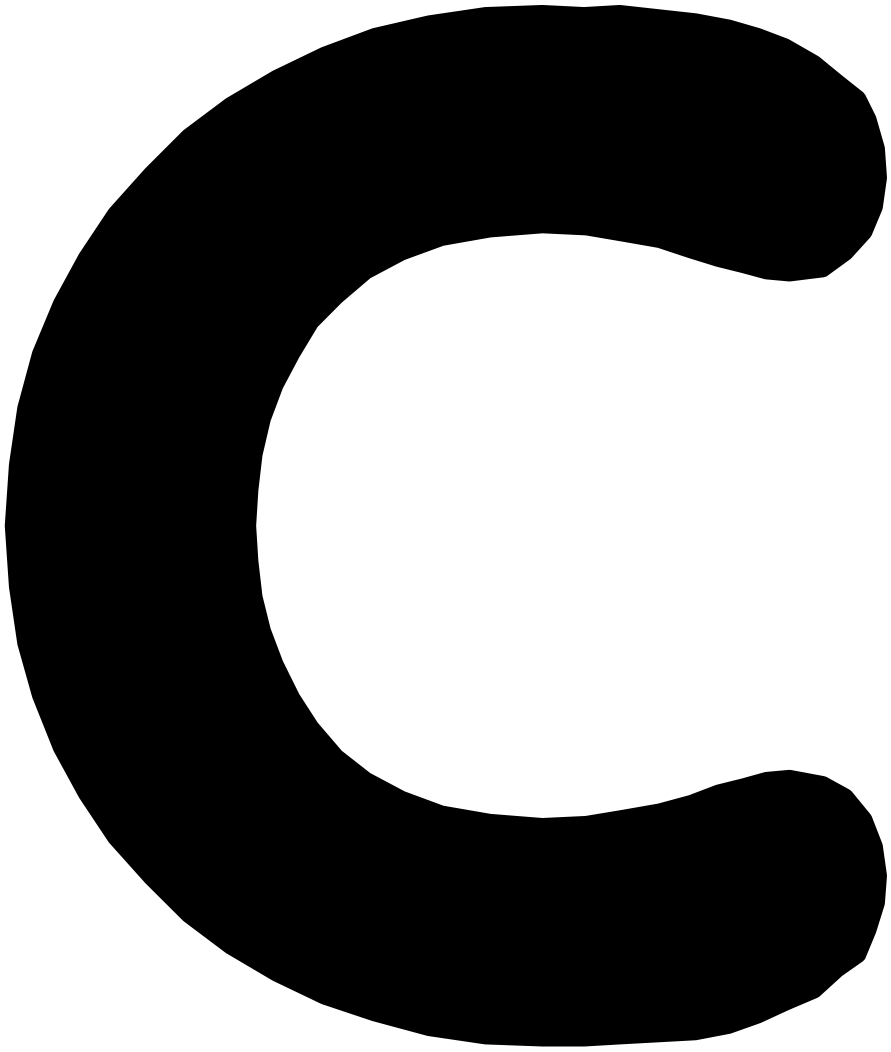
Skriptum

für die Programmiersprache



INHALTSVERZEICHNIS

1. ALLGEMEINES 4

1.1. WOZU PROGRAMMIERSPRACHEN 4
1.2. ZUM LÖSEN VON PROBLEMEN 5
1.3. PROBLEMLÖSUNG SPEZIELL AM COMPUTER 7
1.4. ANALYTISCHES VERFAHREN ZUM PROBLEMLÖSEN (AVP) 11
1.5. PROGRAMMDOKUMENTATION 11
1.5.1. Wartungsdokumentation (Entwicklungsdokumentation) 12
1.5.2. Benutzerdokumentation 12
1.6. ALLGEMEINE EIGENSCHAFTEN EINES GUTEN PROGRAMMES 12

2. ALLGEMEINES ZU C 13

3. ERSTE BEISPIELPROGRAMME 15

3.1. BEISPIELPROGRAMM AM COMPUTER AUSFÜHREN 18

4. RICHTLINIEN ZUR VERNÜNFTIGEN PROGRAMMIERUNG 20

5. VARIABLE, DATENTYPEN UND KONSTANTE IN C 22

5.1. DATENTYP GANZZAHL (INTERNE BINÄRZAHL) 22
5.2. DATENTYP GLEITKOMMA (ENTSPRICHT BINÄREM MASCHINENFORMAT) 23
5.3. DATENTYP CHARACTER (NUR EINE STELLE) 23
5.4. ARRAYS (ZUSAMMENFASSEN MEHRERER ELEMENTE GLEICHEN TYPS) 24
5.4.1. Spezielles zu char - Arrays 26
5.4.2. Suchen von Werten in Arrays 27
5.4.3. Sortieren von Werten in Arrays 29
5.5. BEISPIELE FÜR EIN- AUSGABE VERSCHIEDENER DATENTYPEN 32
5.6. SPEZIELLE I/O FÜR TASTATUR, BILDSCHIRM 36

6. BEFEHLE IN C 38

6.1. DAS LEERE STATEMENT 38
6.2. ZUWEISUNG 38
6.3. WHILE - SCHLEIFE 38
6.4. DO WHILE - DIE „UNTIL“ - SCHLEIFE IN C 40
6.5. FOR - SCHLEIFE (ITERATIVE SCHLEIFE) 40
6.6. IF ELSE (DIE KLASSISCHE BEDINGUNG) 41
6.7. BREAK UND CONTINUE (AUSSTEIGEN AUS SCHLEIFEN) 42
6.8. SWITCH (DIE C VARIANTE DES CASE BEFEHLS) 43
6.9. GOTO (JA AUCH DIESES STATEMENT EXISTIERT) 44
6.10. RETURN (BEENDEN UNTERPROGRAMM, ZURÜCKGEBEN FUNKTIONSWERT) 45

7. DER AUFBAU EINES C - PROGRAMMS 51

7.1. FUNKTIONEN UND UNTERPROGRAMME ALLGEMEIN 51
7.2. SCHNITTSTELLE ZWISCHEN FUNKTIONEN (FUNKTIONSAUFRUF) 53
7.2.1. Parameter 53
7.2.2. Funktionsrückgabewert 54
7.2.3. globale Variable 55
7.3. FUNKTIONEN UND SOURCECODEDATEIEN IN C 55
7.4. LOKALE / GLOBALE VARIABLE (SPEICHERKLASSE, LEBENSDAUER UND GÜLTIGKEITSBEREICH) 58

8. OPERATOREN IN C (DAVON GIBT ES VIELE) 60

9. DATEIEN IN C 67

9.1. GRUNDLEGENDES ZU DATEIEN 67
9.2. STRUKTUREN (ZUSAMMENFASSEN VON ELEMENTEN UNTERSCHIEDLICHEN TYPS) 68
9.3. FUNKTIONEN FÜR DIE DATEIBEARBEITUNG MIT BEISPIELEN 70
9.3.1. Textfiles 71

9.3.2. binäre Files ... *71*
9.3.3. Dateien aus C ansprechen ... *72*
9.3.4. I/O Befehle für Dateien ... *73*
9.3.5. Lesen und Schreiben von Binärfiles ... *74*

10. VARIABLEN- UND FUNKTIONSDEFINITION (EXAKT) ... 83

10.1. TYPATTRIBUTE ... 83
10.2. DATENTYP ... 83
10.2.1. Grundlegende integer Typen ... *84*
10.2.2. Bitfeld Typen ... *84*
10.2.3. aufzählende Typen ... *84*
10.2.4. Gleitkomma Typen ... *85*
10.2.5. Void ... *85*
10.2.6. Arrays ... *85*
10.2.7. Strukturen ... *86*
10.2.8. Unions ... *86*
10.2.9. Pointer ... *87*
10.2.10. Funktionen ... *95*
10.3. KONVERTIERUNGEN ZWISCHEN DATENTYPEN ... 97

11. ANHANG ... 99

11.1. DIE C BIBLIOTHEKEN ... 99
11.1.1. ctype.h ... *99*
11.1.2. math.h ... *100*
11.1.3. signal.h ... *100*
11.1.4. stdio.h ... *100*
11.1.5. stdlib.h ... *102*
11.1.6. string.h ... *102*
11.1.7. time.h ... *103*
11.2. GRAFIKMÖGLICHKEITEN UNTER DOS MIT BORLAND BGI ... 104

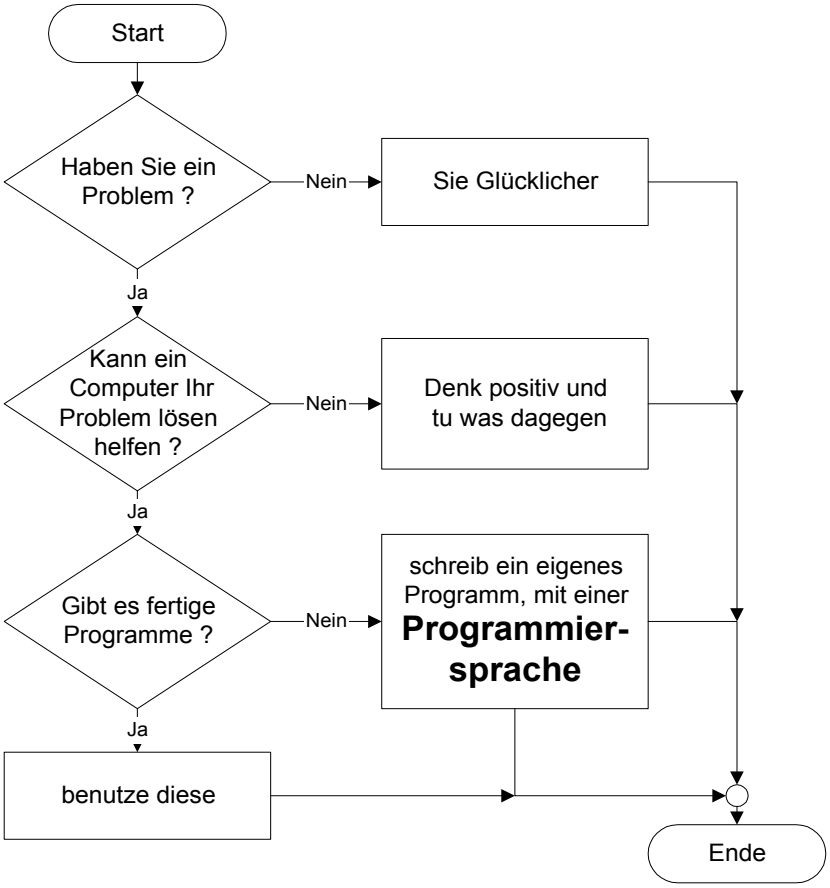
Wie alle Schriftwerke unterliegt diese Skriptum dem Urheberrechtsschutz (beim Vertrieb in den USA ist dafür ein Copyright Vermerk notwendig). Sie können daher für den privaten Gebrauch einige wenige Kopien anfertigen, Sie dürfen diese aber keinesfalls verkaufen und es ist ebenfalls nicht zulässig wenn Sie Teile des Skriptums oder gar das ganze Konvolut als eigene Schöpfung ausgeben.

IMPRESSUM : Herausgeber : Mag. Preißl Johann Pelargonienweg 70, 1220 Wien

1. Allgemeines

1.1. Wozu Programmiersprachen

Diese graphische Darstellungsform nennt man Programmablaufplan.



Vor langer Zeit (bloß 30 Jahre) mußte praktisch jeder programmieren können, der einen Computer bedienen wollte. Heute gibt es eine Unzahl von fertigen Computerprogrammen und viele normale Anwender verwenden Texverarbeitungen oder Computerspiele ohne dafür irgendwelche Programmierkenntnisse zu benötigen.

Beschäftigt man sich jedoch hauptberuflich mit dem Computer, so ist die Programmierung ein wichtiger Teil des dafür nötigen Wissens.

In früherer Zeit mußte man bloß eine Programmiersprache erlernen und konnte sich bereits als Programmierer bezeichnen. Heute sind aber die Anforderungen an Programme und damit ihre Größe wesentlich gestiegen. Auch die Qualität der Programme muß erhöht werden (wessen Computer ist noch nicht abgestürzt).

Programmieren sollte heute eine geplante, ingenieurmäßige Tätigkeit sein

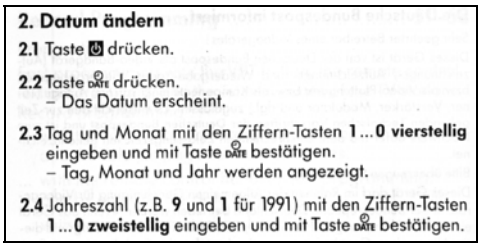
Für eine große Programmieraufgabe (ein komplexes Problem) muß zuerst ein grundsätzliches Lösungsmodell aufgestellt werden, die Gesamtaufgabe ist in kleinere Einheiten (z.B. für mehrere Personen) aufzuteilen und die Einzelteile müssen detailliert vorformuliert werden (z.B. Struktogramm). Jetzt erst wird die Lösung in einer Programmiersprache geschrieben (langläufig als die eigentliche Programmierung bezeichnet). Dann müssen noch die Einzelteile und zum Schluß das/die gesamten Programme gründlich auf korrekte Funktion getestet werden. Alle Arbeitsschritte müssen gut dokumentiert werden, sodaß ein mit der Materie nicht vertrauter Programmierer später Änderungen vornehmen kann.

! Wichtig

1.2. Zum Lösen von Problemen

Vor einem **Problem** stehen wir im allgemeinen dann, wenn uns kein Plan bzw. keine Mittel und Wege bekannt sind, um von einem unerwünschtem Ausgangspunkt zu einem erwünschtem Zielzustand zu gelangen.

Weil nur sehr kleine Probleme auf die Schnelle gelöst werden können, empfiehlt sich ein systematisches Vorgehen. Man wird daher nach einem **Problemlösungsplan** bzw. einem Problemlösungsverfahren suchen. Wird man nicht fündig, dann muß man selbst einen Problemlösungsplan entwickeln.


2. Datum ändern
2.1 Taste ⏻ drücken.
2.2 Taste DATE drücken.
– Das Datum erscheint.
2.3 Tag und Monat mit den Ziffern-Tasten **1**...**0 vierstellig** eingeben und mit Taste DATE bestätigen.
– Tag, Monat und Jahr werden angezeigt.
2.4 Jahreszahl (z.B. **9** und **1** für 1991) mit den Ziffern-Tasten **1**...**0 zweistellig** eingeben und mit Taste DATE bestätigen.

Vor allem Männer sehen es als ernsthaftes Problem an, wenn sie etwas kochen sollen. Normalerweise verwenden sie dann einen Problemlösungsplan, den ein anderer Mensch (wahrscheinlich ein Koch) niedergeschrieben hat - sie suchen das passende Rezept in einem Kochbuch.

Bevor es ShowView gab war es mitunter ein ernsthaftes Problem den Videorecorder zum Aufnehmen einer bestimmten Sendung zu bewegen. Man mußte den zugehörigen Problemlösungsplan - die Bedienungsanleitung gründlich studieren (siehe nebenstehendes Beispiel).

Weitere Beispiele für häufig eingesetzte Problemlösungspläne sind :

- Montageanleitung für Selbstbaumöbel
- Strickanleitung für einen Pullover
- Noten eines Musikstücks
- Stadtplan
- Rechenvorschrift, z.B. Formeln für Volumen, etc.
- Abbildungsvorschrift, z.B. eine Funktion in der Mathematik

Feststellungen zur Problemlösung:

- **Problemlösungspläne** werden **immer von Menschen entworfen**.
- Je nach Problem sind sie für völlig unterschiedliche Zielgruppen konzipiert.
  Denkt der Ausführende selbst mit, so kann die Formulierung Freiräume enthalten. Ist das nicht der Fall, so muß eine sehr detaillierte exakte Verfahrensvorschrift zur Problemlösung formuliert werden.
- Je nach Problem variiert die Darstellung (allgemein verständlicher Text, Text mit Fachausdrücken, Spezialzeichen wie Noten, Formeln, graphische Darstellungen)
- Häufig sind Problemlösungspläne eine Folge von nacheinander auszuführenden Anweisungen.

Einflüsse auf die Problemlösung:

- Erfahrung bzw. Wissen : Viele Übungsaufgaben sind zu Beginn schwer, rückwirkend betrachtet aber gar kein Problem mehr.
- Einstellung: Bin ich interessiert neue Aufgaben zu lösen oder lasse ich sie lieber liegen; kann ich meine Arbeit auch kritisch betrachten.
- Methodik: Anfänger neigen zu einer eher chaotischen Vorgangsweise, die bei Kleinproblemen auch funktionieren kann. Braucht man aber längere Zeit, oder arbeiten gar mehrere Leute an der Problemlösung, dann muß systematisch vorgegangen werden. Selbstverständlich ist auch die passende Idee zur rechten Zeit notwendig, Intuition kann Detailarbeit wohl vermindern aber nicht vollständig ersetzten.

Beispiel Rezept Pichelsteiner Eintopf

Pichelsteiner Topf

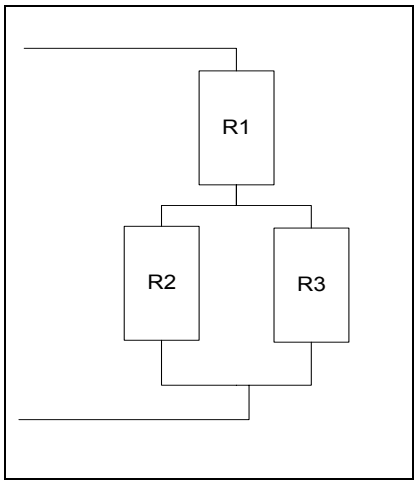
150 g Ochsenmark
300 g Kartoffeln
1 kl. Kopf Wirsing
200 g Mohrrüben
einige Champignons
100 g grüne Bohnen
Petersilienwurzel
1/2 Blumenkohl
1 Zwiebel
Salz, Pfeffer
500 g Kalb-, Hammel- und Schweinefleisch gemischt und gut abgehangen
1/4–3/8 l Wasser oder Brühe

Nehmen Sie einen passenden Topf und arrangieren Sie lagenweise. Den Boden bedecken Sie mit dem in Scheiben geschnittenen Rindermark, dann kommt die Hälfte des kleingeschnittenen Gemüses und der Kartoffeln darauf, das würfelig geschnittene, gesalzene und gepfefferte Fleisch und zum guten Schluß der Rest des Gemüses und der Kartoffeln. Begießen Sie es mit 1/4–3/8 l Brühe, verschließen Sie den Topf gut, damit der Dampf nicht entweichen kann, und lassen Sie auf kleiner Flamme alles weichdünsten.
Das wird etwa 1 1/2 Stunde dauern, und ich würde Ihnen raten, nicht umzurühren, höchstens mal den Topf zu rütteln.

Das nebenstehende Rezept ist vor allem wegen seiner vielen Zutaten (bei Eintöpfen häufig) abgebildet.
Die Problemlösung selbst ist eine Reihe von knapp gehaltenen, in Sätze gekleideten Anweisungen. Jede Anweisung besteht aus mehr oder minder elementaren Handlungen, bei denen meistens Zutaten verwendet bzw. verarbeitet werden.

Jede Zutat gehört zu einer bestimmten Art; Kartoffel, Wirsingkohl oder Mohrrüben sind Gemüse, Salz und Pfeffer sind Gewürze.

Wie elementar (genau) eine bestimmte Handlung formuliert werden muß, hängt ganz wesentlich vom Ausführenden ab. Ein Koch wird mit noch knapperen Formulierungen auskommen, ein Küchenlaie wird wohl keinen „passenden Topf" finden. Elementar ist eine Handlung dann, wenn sie nicht mehr sinnvoll zerlegt werden kann.



Beispiel Widerstandsberechnung

Bei der Berechnung seriell und parallel geschalteter Widerstände kommt in diesem Fall folgende Vorgangsweise zur Anwendung: Seriell (hintereinander liegende) Widerstände werden addiert, bei parallelen (nebeneinander liegenden) Widerständen muß deren Produkt durch deren Summe dividiert werden. Viel besser verständlich ist hier die Schreibweise als Formel .

$$R = R1 + \frac{R2 * R3}{R2 + R3}$$

wenn R1 = 10 Ω
R2 = 8 Ω
R3 = 12 Ω

$$R = 10 + \frac{8 * 12}{8 + 12}$$

Auf einem Taschenrechner würde die getippte Anweisungsfolge so aussehen:

10 + (8 x 12) / (8 + 12) =

Die nachfolgende Tabelle soll zeigen, welche Begriffe in welcher Problemumgebung verwendet werden (Begriffe in einer Zeile haben jeweils eine ähnliche Bedeutung)

	beim Pichelsteiner Eintopf	bei der Widerstands-berechnung	als Programmierer bzw. Informatiker
Ausführender ist	Mensch	Taschenrechner	Computer
der Problemlö-sungsplan heißt	Rezept	Formel	Algorithmus
Anweisung	„gesalzenes Fleisch in den Topf geben"	Formelteile (8+12)	Befehle einer Programmier-sprache / eines Programms
elementare Hand-lungen sind	„Fleisch salzen"	Operationen (+)	Einzelbefehle (X = X + 1) , Operationen für das Objekt
Anweisungen durchgeführt mit	Zutat	Zahlen (8,10,12)	Objekte (bei Programmen Variable, allgemein Daten)
zusammengefaßt zu	Zutatenart	Reelle Zahlen	Objektart

Übungen: Verfassen Sie Problemlösungspläne für die folgenden Probleme

- Sie sollen einem nicht ortskundigen Freund Ihren Schulweg erklären. Mit Ihren schriftlichen Aufzeichnungen soll dieser möglichst schnell zur Schule gelangen
- Kaffee kochen mit einem Filterautomat
- Mit einem Auto wegfahren
- Tanken bei einer Selbstbedienungstankstelle

1.3. Problemlösung speziell am Computer

Problemlöser ist der Computer

Wenn ein Computer die Problemlösung ausführt, so muß der Problemlösungsplan nicht nur sinnvoll und richtig sein, er muß auch bis auf den letzten Beistrich exakt und genau formuliert werden - noch dazu in einer Programmiersprache. Schon bei kleinen Fehlern lehnt der Computer die Problemlösung ganz ab oder macht unvorhergesehene Dinge. Das Erlernen einer Programmiersprache ist Gott sei Dank wesentlich einfacher als beispielsweise Französisch zu lernen. Beherrscht man die Sprache einigermaßen, muß man sich auch Gedanken über die sinnvolle Anwendung machen. Schließlich könnte man ja auch im Deutschen grammatikalisch richtige aber inhaltlich vollkommen sinnlose Sätze bilden. Ähnlich verhält es sich bei Programmiersprachen.

Als **Syntax** einer Sprache bezeichnet man die richtige Schreibweise der Wörter, die korrekte Setzung von Sonderzeichen und die „grammatikalisch" richtige Anordnung. Schreiben Sie ein Programm in einer Programmiersprache, so wird die Syntax Ihrer Zeilen vom Computer überprüft.

Als **Semantik** bezeichnet man die inhaltliche Bedeutung bzw. die Logik Ihres Programms. Ob Ihr Programm also auch genau das tut, was es soll, muß leider von Ihnen selbst überprüft werden - dies ist auch die Haupttätigkeit beim Testen von Programmen.

der Algorithmus ist das Verfahren der Problemlösung

Der Problemlösungsplan bzw. Teile desselben wird bei Computern Algorithmus genannt. Ein **Algorithmus** ist eine Verfahrensvorschrift, die so genau formalisiert ist, daß sie von einem Computer (dessen Prozessor) ausgeführt werden kann. Der Algorithmus besteht aus Anweisungen, die wiederum Operationen mit verschiedenen Objekten ausführen. Ein Algorithmus bildet Eingabeobjekte auf Ausgabeobjekte ab. Er beschreibt in welchen Anweisungen (Operationen) die Eingabeobjekte verwendet werden um die Ausgabeobjekte zu erzeugen.

Man kann einen Algorithmus mit verschiedenen Notationen darstellen:

- Pseudocode: Verbale Darstellung, meist angelehnt an die Schreibweise von Programmiersprachen. Ein Rezept (Pichelsteiner Eintopf) wäre ein Pseudocode - Beispiel.
- Programmablaufplan: Ältere graphische Form (DIN Norm 66001). Die Darstellung „wozu Programmiersprachen" ist mit Programmablaufplansymbolen gestaltet.

Struktogramme stellen Algorithmen graphisch dar

- **Struktogramm**: Die günstigste Darstellungsform für Algorithmen ist das Struktogramm. (DIN Norm 66261) Durch die graphische Darstellung ist gute Lesbarkeit gesichert. Die graphischen Symbole ermöglichen keine chaotische Vorgangsweise, sondern führen automatisch zu gut **strukturierten Programmen**, welche heute Stand der Technik sind .

Beispiel: Widerstandsberechnung als Struktogramm. Sinnvollerweise wird man gleich die allgemeine Formel (mit R1,R2,R3) verwenden, damit man für verschiedene Ohm-Eingabewerte den Gesamtwiderstand R errechnen kann.

Algorithmus Widerstandsschaltung

verwendete Objekte:
Variable R, R1, R2, R3

lies die Werte für R1, R2, R3 (von Tastatur)
berechne $R = R1 + \frac{R2 * R3}{R2 + R3}$
schreibe den Wert von R auf den Bildschirm

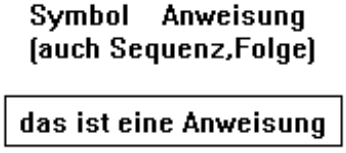
Variable sind die normalen Objekte innerhalb eines Programms

Eine **Variable** ist ein Objekt mit einem vom Programmierer vergebenen Bezeichner (dem Namen z.B. R1, R2). Eine Variable hat einen Wert, der durch Anweisungen bzw. Operationen verändert werden kann (also variabel ist).

Das obige Beispiel enthält 3 Anweisungen, wobei in der ersten Anweisung drei Zahlenwerte vom Benutzer mit der Tastatur eingegeben und in die Variablen R1,R2,R3 hineingestellt werden. In der zweiten Anweisung wird eine umfangreiche Berechnung mit mehreren Operationen durchgeführt und das Ergebnis in die Variable R gestellt. In der dritten Anweisung nimmt man den Wert der Variablen R und schreibt ihn auf den Bildschirm.

Struktogrammelement **Anweisung**

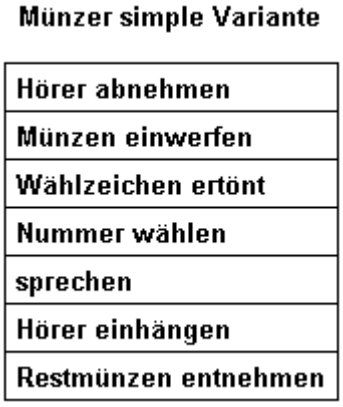
In den bisherigen Beispielen konnte man den Problemlösungsplan stets als eine Folge von einmal hintereinander auszuführenden Anweisungen darstellen. Die obige Darstellung besteht daher auch nur aus 3 hintereinander ausführbaren Anweisungen, die als rechteckige Kästchen dargestellt sind. Dieses Symbol (für die einzelne **Anweisung** bzw. für einen Verarbeitungsschritt) nennt man auch **Sequenz** oder **Folge**, weil man meist mehrere Kästchen untereinander anordnet und auch exakt in dieser Reihenfolge abarbeitet.



Beispiel: Bedienung eines Münzfernsprechers, zuerst eine simple Version , bei der einzelne Anweisungen hintereinander ausgeführt werden.

Man spricht in jedem Fall, egal ob sich der Gesprächspartner meldet oder nicht.

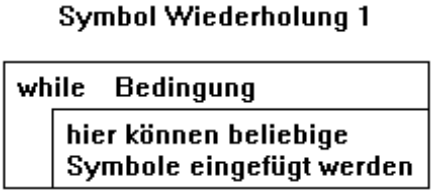
Man sollte eine sinnvollere Variante entwerfen, bei der man mehrere Wählversuche machen kann. Dafür benötigt man aber Symbole für die wiederholte Durchführung



Manche Probleme lassen sich aber nicht mittels einer einfachen Reihenfolge lösen. Beispielsweise werden einzelne Anweisungen nur dann ausgeführt, wenn bestimmte Bedingungen zutreffen. Umgangssprachlich formuliert sagt man „Wenn Bedingung, dann Anweisung“, oder auch „Wenn Bedingung, dann Anweisung-1 sonst Anweisung-2“. Auch ist es machmal sinnvoll einzelne Anweisungen mehrfach zu wiederholen. Dazu schreibt man diese nicht mehrere Male hintereinander, sondern sagt sinngemäß „solange Bedingung erfüllt wiederhole Anweisung“.

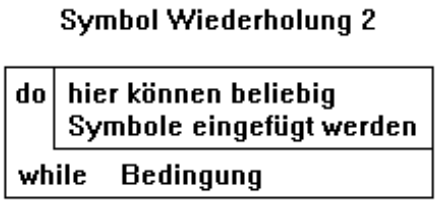
Struktogrammelement **Wiederholung**

Benötigt man Wiederholungen, dann muß man die Symbole für die Wiederholung verwenden.



Dabei werden Anweisungen (stehen dort wo man beliebige Symbole einfügen kann) in Abhängigkeit von einer Bedingung wiederholt ausgeführt.

Die Bedingung wird dabei überprüft ob sie wahr oder falsch ist. Solange (while) die Bedingung wahr ist, werden die im Symbol enthaltenen Anweisungen ausgeführt. Beim Symbol Wiederholung 2 schreibt man statt while in anderen Sprachen auch until. Do until bedeutet führe die Anweisungen aus bis (until) die Bedingung wahr wird.

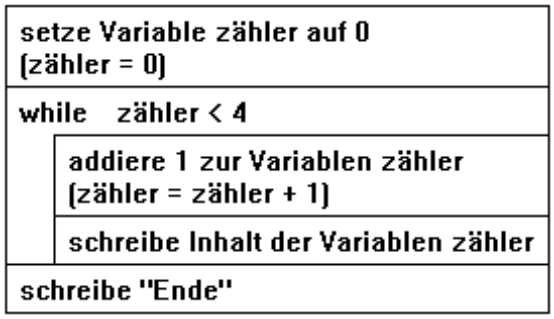


	Wiederholungssymbol 1	Wiederholungssymbol 2
nennt man	**While** - Schleife abweisende Schleife kopfgesteuerte Schleife	**Do while** - Schleife nicht abweisende Schleife fußgesteuerte Schleife in anderen Sprachen auch do until
Unterschied	die Bedingung wird jeweils vor den Anweisungen geprüft ➔ 0 bis n Durchläufe	die Bedingung wird jeweils nach den Anweisungen geprüft ➔ 1 bis n Durchläufe

Beispiel : um den Ablauf besser darzustellen folgt hier ein kleines Beispiel

einfaches Zählbeispiel
Variable : zähler

Ausgabe auf dem Bildschirm :
1
2
3
4
Ende


setze Variable zähler auf 0
(zähler = 0)
while zähler < 4
addiere 1 zur Variablen zähler
(zähler = zähler + 1)
schreibe Inhalt der Variablen zähler
schreibe "Ende"

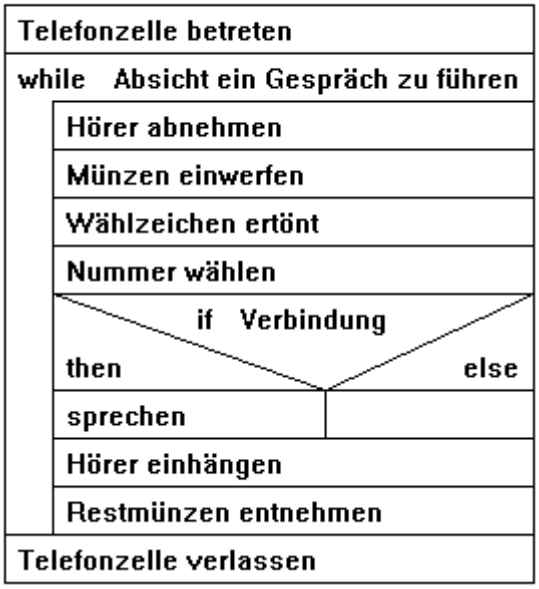
Dieses Struktogramm setzt sich aus mehreren Symbolen zusammen. Die Abarbeitung beginnt wie immer beim obersten Symbol und endet beim untersten. Das Symbol Wiederholung enthält zwei einzelne Anweisungen, die im konkreten Fall viermal ausgeführt werden, weil die vorher überprüfte Bedingung (zähler kleiner 4) nur viermal wahr ist. Dann hat zähler den Wert 4, die Bedingung ist falsch und die Verarbeitung geht nach dem Wiederholungssymbol weiter.

Münzer sinnvolle Variante

Bei dieser Version des Münzfernsprechers kann man wiederholt telefonieren.

Die vorher dargestellte simple Variante wurde komplett in eine Schleife (ein Wiederholungssymbol) hineingestellt und kann somit öfter wiederholt werden.

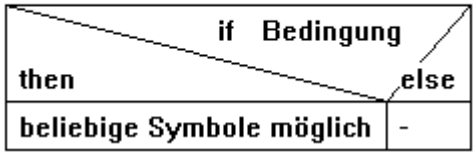
Nach der Anweisung „Nummer wählen“ ist als neues Symbol die Auswahl verwendet worden. Wenn (if) eine Verbindung zustandekommt, dann (then) sprechen, ansonsten (else) nichts tun. Nach dieser Auswahl geht es in jedem Fall mit „Hörer einhängen“ weiter.



Struktogrammelement
Auswahl

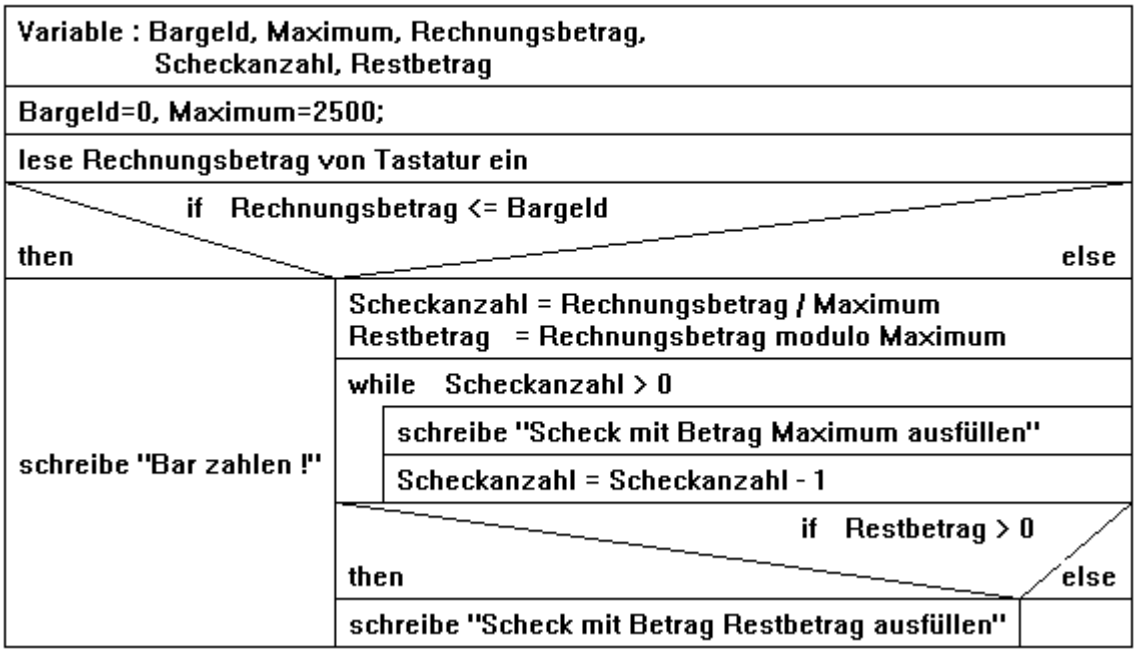
Das Symbol Auswahl wird dazu verwendet um abhängig von einer Bedingung Anweisungen auszuführen oder zu überspringen. Wenn auch rechts beim else Anweisungen stehen, dann wird entweder der then Zweig oder der else Zweig bearbeitet.

Symbol Auswahl



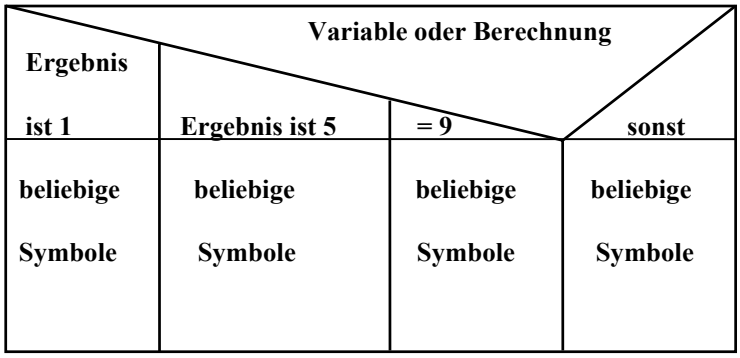
Das nächste Beispiel zeigt vor allem, wie verschiedene Struktogrammsymbole ineinander geschachtelt werden können.

Programm zum Bezahlen einer Rechnung mit Schecks oder bar



Als Spezialform der Auswahl gibt es noch die sogenannte Fallunterscheidung. Dabei wird abhängig vom Inhalt einer Variablen oder einer Berechnung eine von mehreren Anweisungen durchgeführt.

Struktogrammelement
Fallunterscheidung



Übungen :

- Erweitern Sie das Münzfernsprecherstruktogramm so, daß ein kaputtes Telefon (z.B. Hörer fehlt) oder eine sogenannte tote Leitung (kein Wählzeichen) erkannt werden.
- Erstellen Sie ein Struktogramm für den Arbeitstag einer Supermarktkassiererin
- Erstellen Sie ein Struktogramm, welches eine Zahl einliest , daraus die Fakultät der Zahl (n!) berechnet und ausgibt.
- Lesen Sie 2 Zahlen ein und geben Sie den Wert der größten Zahl aus.
- Lesen Sie 3 Zahlen ein und geben Sie den Wert der größten Zahl aus.
- Lesen Sie einen Bruttogehalt ein und ermitteln Sie den abzuführenden Steuerbetrag (grob vereinfacht) unter 10000,- S steuerfrei, von 10000,- S bis 20000,- S 20 % Steuer, über 20000,- S 30 % Steuer.
- Lesen Sie 3 Zahlen ein und geben Sie die Summe und den Durchschnitt aus.
- Lesen Sie eine Folge von Zahlen ein; wenn die Zahl 0 eingelesen wird geben Sie Summe und Durchschnitt aller Zahlen aus und beenden das Programm.

1.4. Analytisches Verfahren zum Problemlösen (AVP)

Dies hier ist eine mögliche Vorgangsweise, die dem Anspruch auf eine ingenieurmäßige Softwareentwicklung gerecht werden kann.

Wie schon häufig betont wurde, muß bei größeren Programmen/Problemstellungen systematisch vorgegangen werden. Hier wird ein möglicher Weg gezeigt, wie das passieren könnte:

1. Wie lautet das Problem
   Man überlege sich eine passende prägnante **Problembenennung** als geistigen „Aufhänger". Überlegungen wie „Gibt es ein vergleichbares Problem?" oder „Wie kann ich das Problem verallgemeinern?" sind sicher hilfreich.

2. Was ist bekannt?
   hier sind alle bekannten Eingangsgrößen (Programmeingabe)aufzuzählen

3. Was wird gesucht?
   Aufzählung aller Ausgangsgrößen (Programmausgabe)

4. Spezifizierung (genaue Beschreibung)
   - der Aufgabenstellung; wenn nötig Unterteilung in Teilaufgaben
   - der Eingangs- und Ausgangsgrößen (jeder Teilaufgabe)

   Bei Teilaufgaben werden die Punkte 5 bis 7 (oder auch 5 - 8) pro Teilaufgabe durchgeführt.

5. Suchen der Problemlösung - des Algorithmus
   man verwendet entweder einen bewährten Algorithmus
   bzw. entwickelt gemäß Aufgabenstellung einen neuen.
   Darstellung desselben in Form von Struktogrammen bzw. Pseudocode

6. Überprüfung der Korrektheit des Algorithmus mittels Schreibtischtest - man spielt den Ablauf des Algorithmus mit Testeingabegrößen (Testdaten) durch und korrigiert allfällige Mängel.

7. Codierung - Übersetzen (umschreiben) des Algorithmus vom Struktogramm in die Programmiersprache.

8. Testen des Programms auf korrekte Funktion. Ein Testprotokoll kann Ihre Testtätigkeit beweisen (z.B. Hardcopys vom Bildschirm machen). Bedenken Sie, daß Programmierer bereits gerichtlich zu empfindlichen Strafen verurteilt wurden, weil sie wichtige Programmteile (konkret der else - Zweig einer Auswahl) nicht getestet haben. Geschehen bei einem Programm zur Steuerung von Verkehrsampeln in Amerika.

9. Zusammenfassen der schriftlichen Unterlagen aus allen bisherigen Punkten zur Programmdokumentation (auch Wartungsdokumentation genannt). Muß später ein anderer Programmierer Ihr Programm ändern bzw. weiterentwickeln, dann wird er hoffentlich Ihre Doku schätzen. Auch Lehrer schätzen gute Dokumentationen.

10. Benutzerhandbuch (Dokumentation für den Anwender des Programms) fertigstellen. Hier wird zu Beginn Sinn und Zweck des Programms beschrieben (ähnlich zu den Punkten 1 - 3). Erst dann soll die konkrete Bedienung des Programms geschildert werden. Hinweise auf notwendige Computer Hard- und Softwareausstattung komplettieren die Benutzerdokumentation.

1.5. Programmdokumentation

Große Teile der Dokumentation liegen bereits in Form der Entwicklungsdokumentation (= AVP) vor. Die volle Programmdokumentation besteht aus zwei Teilen :

1.5.1. Wartungsdokumentation (Entwicklungsdokumentation)

Sie ermöglicht einem nachfolgenden Programmierer das detaillierte Kennenlernen des Programmes, wenn Änderungen oder Ergänzungen notwendig werden oder zur Behebung von Fehlern, die erst später auftreten. **Sie enthält :**

- Name des Programmes und des Programmierers, Datum dieser Fassung
- Überblick über das Programm, Aufgabenstellung
- Problemanalyse, Beschreibung der Algorithmen
- Beschreibung der Unterprogramme (Aufgabe, Parameter, Rückgabewert)
- Variablenliste (Datentyp, Wertebereiche, Beschreibung)
- Ausgaben auf Bildschirm und Drucker; Masken, Druckbilder
- Eingaben und sinnvolle Werte
- graphische Programmdarstellung (z.B. Struktogramm)
- Testdaten und Ergebnisse, Testlaufprotokoll, Schreibtischtest
- Programmcode mit Kommentaren

1.5.2. Benutzerdokumentation

Sie soll alle Fragen, die dem Benutzer vor und während der Arbeit mit dem Programm kommen, beantworten können. Sie wird in normalen, deutschen Sätzen, nicht in Computer-slang, abgefaßt. **Folgende Punkte sollten enthalten sein :**

- Überschrift mit kurzer Erklärung des Programmes
- Installation des Programmes (Hardware- u. Softwareanforderungen)
- Start des Programmes
- Bedienungsanleitung (Ein/Ausgaben, wichtige Tasten)
- Verhalten im Fehlerfall
- Weitere wichtige Informationen (z.B. Einschränkungen)

1.6. Allgemeine Eigenschaften eines guten Programmes

1. **aus der Sicht des Anwenders :**

- Das Programm muß richtige Ergebnisse liefern und absturzfrei laufen
- Führung des Benutzers mit Menü
- Strukturell gleich aufgebaute Bildschirmseiten (Gleiches an gleicher Stelle, z.B. Meldungszeile)
- Erklärende Texte bei Eingaben und Ausgaben von Werten
- Eingabeüberprüfungen
- Genaue Fehlermeldungen (nicht: Fehler, besser: Datei nicht gefunden, noch besser: Benutzeranleitung)
- Hilfetexte, die immer mit derselben Taste (meist F1) erreichbar sind

2. **aus der Sicht der Wartung :**

- Programmierrichtlinien beachten (Programmaufbau, Konstantenvereinbarungen z.B. #define GROSS 10,...)
- Code strukturiert schreiben (einrücken, { und } untereinander,...)
- Sprechende Namen verwenden
- Ausführliche Kommentare, z.B. bei schwierigen Algorithmen

2. Allgemeines zu C

C ist eine universelle Sprache für komplexere Aufgaben

Die Programmiersprache C wurde 1972 von Dennis M. Ritchie bei den Bell Laboratories (AT&T) entwickelt. Zusammen mit Ken Thompson arbeitete Ritchie damals daran, das eben entwickelte Betriebssystem Unix auf einen weiteren Prozessor zu portieren. Um den großen Aufwand durch das Neuschreiben des Betriebssystems in der Maschinensprache des jeweiligen Prozessors zu vermeiden, versuchte man eine Programmiersprache zu entwickeln, in der das Betriebssystem geschrieben werden konnte. Der Portierungsaufwand reduziert sich dann auf das Anpassen des Betriebsystemkerns und des Compilers. Nach zwei Probeversuchen (BCPL und B) schien der dritte Versuch brauchbar. So entstand C als Sprache, in der über 90 % von Unix geschrieben ist und die offensichtlich zum Schreiben von Betriebssystemen und anderer umfangreicher Software tauglich ist.

In C wurden mehr Codezeilen geschrieben als in irgendeiner anderen Sprache.

C ist eine international genormte Sprache

Von 1983 bis Dezember 1989 wurde von der Ansi Normungskommision die Normierung von C vorangetrieben. Als Ergebnis liegt "Ansi C" (Ansi X3.159-1989) vor, das heute von allen Compilern unterstützt wird. Ansi C ist inzwischen auch eine internationale Norm (ISO /IEC 9899:1990). Ältere Programme, die vor dem Vorliegen der Norm entstanden, sind aber nicht in irgendeinem Dialekt geschrieben, sondern im sogenanntem K&R (Kerninghan und Ritchie) C. Durch das Buch "Programmieren in C" haben sie nicht nur die Syntax, sondern auch den Stil von C Programmen vorgegeben, sodaß zwischen genormten und früheren Programmen nur geringe Unterschiede bestehen. Wesentliche Unterschiede bei verschiedenen C Compilern ergeben sich hauptsächlich bei Compiler- oder maschinenspezifischen Erweiterungen, die in der Norm nicht enthalten sind. Nicht genormt sind typischerweise Graphikausgabe, maschinenspezifische Ausgabe (Bildschirmspeicher), Window Systeme, hardwarenahe File I/O.

C++ ist die objektorientierte Aufwärtsentwicklung von C

Als Weiterentwicklung von C gewinnt C++ zunehmend an Bedeutung. Weil aber C praktisch vollständig in C++ enthalten ist, ist es sinnvoll, zuerst C zu erlernen. Sobald man einige Übung hat und größere C Programme schreiben will bzw. soll, ist der Umstieg auf C++ sinnvoll. Auch die meisten heutigen Compiler sind C++ Compiler, sie können C und natürlich auch C++ Programme übersetzen..

Heute kann man C (bzw. C++) als die wichtigste Programmiersprache für Systementwickler bezeichnen; tauglich für alle komplexen Programmieraufgaben, speziell auf kleineren und mittleren Systemen. Obwohl Compiler auch auf Großanlagen existieren, wird sich, bedingt durch die dortige Rechnerarchitektur (blockorientierte Bildschirme, etc.), die Sprache langsamer durchsetzen. Der Großteil aller auf PCs und Unix Maschinen angebotener Software ist in C geschrieben. Beispielsweise bezeichnet sich die Datenbank Oracle als das größte C Programm der Welt. Programme, die auf verschiedenen Systemen mit guter Performance laufen sollen, kann man praktisch nur in C bzw. C++ schreiben.

C ermöglicht nicht nur flexible Programmierung sondern auch viele Fehler

! Wichtig

Allerdings ist es in C auch besonders einfach, schlimme Fehler zu machen. Die ersten größeren Programme von C Anfängern kommen oftmals nicht zum Laufen, weil die jeweiligen Entwickler zu chaotisch vorgehen. Neben der üblicherweise existierenden Möglichkeit, Tabellenindices ins Nirwana zu setzen, kann man dies in C auch mit Pointern tun. Auch kann man relativ leicht Variableninhalte mit deren Adressen verwechseln. In all diesen Fällen vernichtet man irgendwo im Hauptspeicher einige Bytes, die man später (an einer völlig anderen Programmstelle) dringend brauchen würde. Man tut daher gut daran, C Programme seriös zu entwickeln und in kleinen Einheiten auszutesten. Auch sollte man Listprogramme und andere einfache Anwendungen nicht unbedingt in C schreiben, sondern solche Probleme besser und schneller mit speziell dafür geschaffener Software (meist der 4.Generation) lösen.

C ist sowohl maschinennahe, verfügt aber auch über alle Eigenschaften einer höheren Programmiersprache (der 3. Generation). Das wären im Detail:

- Datentypen:
  skalare Typen (integer,float,char)
  Strukturen
  Arrays

- Darstellungsmöglichkeit komplexer Typen (Listen, Bäume)

- Kontrollstrukturen (zur Strukturierung):
  Programmblöcke (mit {})
  if
  while
  do while(until)
  case
  for (iterative Schleife)

- Methoden zur dynamischen Hauptspeicherverwaltung:
  Pointer
  Hauptspeicheranforderung zur Laufzeit (malloc)

- Modularität (Funktionen, Unterprogramme)

- Precompiler (Macromöglichkeit)

C verfügt über umfangreiche Funktionsbibliotheken

Neben dem Sprachumfang selbst, der sehr klein gehalten ist, gibt es zu jedem Compiler noch eine umfangreiche, großteils genormte Bibliothek vorgefertigter Funktionen, die wesentlich zur Brauchbarkeit von C beiträgt. Dort befinden sich sämtliche Möglichkeiten der Ein- und Ausgabe (keine eigenen Sprachbefehle), alle mathematischen Funktionen, Möglichkeiten zur Bildschirmansteuerung, etc. Auch alle speziellen Zugangsmöglichkeiten zum Prozessor der Maschine, die in den anderen Hochsprachen fehlen, sind durch Funktionen realisiert.

Erwirbt man irgendwelche Produkte (z.B. Window Systeme, Isam Datei Management), so werden auch diese als Funktionsbibliothek geliefert. Beispielsweise ist die Programmierung von MS-Windows aus C mittels vieler Funktionen realisiert.

Als klassische Programmierumgebung zu C gab es neben dem Compiler cc und dem Linker ld auch noch weitere Werkzeuge, wie lint (Syntaxprüfung), cb (Beautyfier), ar (Bibliotheken), sdb (Debugger), cflow (Aufrufhierarchie), make (automatische Compilierung), etc. Diese Werkzeuge sind wie der Compiler Bestandteile des Unix Betriebssystems. Viele andere Computerhersteller unterstützen C auf ihren Maschinen und bieten ebenfalls Compiler an.

gute und preisgünstige Compiler auf PC's

Auf PCs ist C eine der wichtigsten Sprachen und kann durch Produkte wie Visual C++, Borland C++ oder Symantec C++ sehr effizient eingesetzt werden. Wie man sieht, können diese Compiler nicht nur C sondern auch C++ Programme übersetzen. Im Gegensatz zur traditionellen Unix Umgebung ist es am PC üblich, mit Full-Screen orientierten, menügeführten und mit umfangreichen Hilfe-Systemen versehenen Produkten zu arbeiten.

Aus der zahllosen angebotenen C-Literatur werden hier nur zwei Bücher erwähnt:

- die C-Bibel vom Meister: Kerninghan, Ritchie "Programmieren in C" Hanser

- Darstellung der Norm: Plaugher, Brody "Ansi und ISO Standard C Programmer's Ref." Microsoft Press

Neben diesen, eher für Fortgeschrittene gedachten Werke, gibt es noch viele Bücher, welche auch die Programmierumgebung der jeweiligen Compiler und deren Spracherweiterungen näher beschreiben.

3. Erste Beispielprogramme

Struktogramm :

Minibeispiel
Variable : keine
schreibe "Hallo Leser CRLF"

Jede Programmiersprache hat eine feste Syntax (Schreibweise), die eingehalten werden muß, damit der Computer (genauer der Compiler der Programmiersprache) unser Geschreibsel auch verstehen kann. In C gibt es häufig geschwungene Klammern und Strichpunkte. Unser Struktogramm, welches nur einen Schreibbefehl enthält, wird trotzdem in mehrere Zeilen C-Programm umgesetzt:

```
#include <stdio.h>
void main()
{
   printf("Hallo Leser\n");
}
```

„#include <stdio.h>" Wenn man Schreibbefehle verwendet, dann muß am Programmbeginn diese Zeile stehen. (Es wird Information für die Ausführung von Schreibbefehlen eingefügt)

„void main ()" Funktion namens „main", beginnt hier mit { und endet mit }. void und () besagen, daß die Funktion keine Daten zurückgibt und auch keine erhält. Jeder Teil Ihres Struktogramms wird im Programm zu einer eigenen Funktion, das Programm beginnt seinen Ablauf immer mit der Funktion main. Deshalb ist auch der Name main zwingend, während andere Funktionen beliebig heißen können.

„{ ..... }" Geschwungene Klammern fassen mehrere Befehle zu einer Einheit (einem Programmblock) zusammen. Hier sind das jene Befehle, die zur Funktion main gehören

„printf ("....\n");" Schreibbefehl, schreibt .... auf den Bildschirm und bewegt den Cursor in die nächste Zeile (durch \n). CRLF bedeutet Carriage return / LineFeed. Alle Befehle müssen mit ; beendet werden.

Hätte Ihr Struktogramm gar keine Anweisung, dann würde daraus in C das folgende Programm, welches natürlich auch nichts tut.

```
void main()
{
}
```

Die eigentlichen Anweisungen werden zwischen den geschwungenen Klammern geschrieben. Sie können praktisch jedes Struktogrammsymbol in einen entsprechenden C-Befehl umsetzen. Am Beginn werden Informationen für das gesamte Programm eingefügt (wie z.B. #include <...>). Am Ende können weitere Funktionen angehängt werden.

Struktogramm :

Auswahlbeispiel	
Variable : zeichen	
lese zeichen	
if (zeichen = 'A')	
then	else
schreibe "Es ist ein A CRLF"	schreibe "nein kein A CRLF"

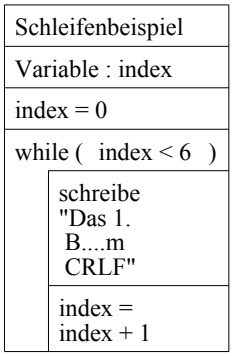
```
#include <stdio.h>
void main()
{
   char zeichen;
   zeichen = getchar();
   if ( zeichen == 'A')
        printf ("Es ist ein A\n");
   else printf ("nein kein A\n");
}
```

„char zeichen" In diesem Beispiel gibt es eine Variable namens zeichen vom Typ char (Objektart Zeichen)

„zeichen=getchar()" In diese Variable wird mit dem Befehl getchar() ein Zeichen (z.B. ein Buchstabe) von der Tastatur eingelesen. Ein = bei Wertzuweisungen.

„if (zeichen=='A') " Auswahl (if) mit Bedingung in der Auswahl. Ein == beim Vergleich.

Struktogramm :




```
/* Beispielprogramm  Kommentar zwischen Schrägstrich Stern             */
#include <stdlib.h>                /* Preprocessor Anweisungen,        */
#include <stdio.h>                 /* kopieren Sourcecode ins Programm */
#define  SCHLEIFENZAHL 6           /* ersetzt im restlichen Programm   */
                                   /* " SCHLEIFENZAHL " durch "6"      */
void main()                        /* Funktion main (keine Parameter)  */
{                                  /* Beginn Programmblock             */
   int index;                      /* Variable index vom Typ integer
                                             Ende mehrzeilger Kommentar */
   index = 0;
   while (index < SCHLEIFENZAHL)             /* Wiederholung          */
   {   printf("Das 1. Beispielprogramm.\n"); /* Ausgabe einer Zeile    */
       index = index + 1;                    /* index erhöhen !!       */
   }                                      /* Ende Programmblock } while */
}                                         /* Ende Programmblock   main  */
           /* hier ist nur Kommentar oder eine neue Funktion sinnvoll */
```

Dieses Programm enthält jede Menge Kommentar. Jeder Text, der zwischen /* und */ steht ist ein Kommentar und dient nur dem Autor oder Leser zum besseren Programmverständnis. Er könnte weggelassen werden. In C++ ist auch alles Kommentar, was zwischen // und dem Zeilenende steht.

der Kopfteil dieses C-Programms

Zu Beginn gibt es sogenannte Preprocessoranweisungen, die mit # beginnen und noch in derselben Zeile enden. Ein #include kopiert Sourcecode (ergänzenden Programmtext) ins Programm. Immer wenn man Befehle aus vordefinierten Bibliotheken verwendet (printf) muß man auch einen dazugehörigen #include im Programm haben.
Ein #define erzeugt sogenannte symbolische Konstante. Der Text "SCHLEIFENZAHL" ist hier ein symbolischer Name für den Text "6", wo immer “SCHLEIFENZAHL“ im Programm vorkommt wird es durch 6 ersetzt. Später noch benötigte vordefinierte symbolische Konstante sind z.B. EOF oder NULL, welche beide innerhalb von stdio.h definiert sind. Mit #include werden diese ins Programm kopiert und sind verfügbar. An dieser Stelle im Programm finden sich später noch Prototypen eigener Funktionen sowie globale Variable, doch dazu später.

das eigentliche Programm

Jedes C Programm muß genau eine Funktion mit Namen main beinhalten, die beim Programmstart aufgerufen wird. Das ist jener Teil des Algorithmus, mit dem das Programm beginnen soll. Bestand der Algorithmus nur aus einem Teil (wie hier), dann stellt die Funktion main auch schon das ganze Programm dar. Gibt es noch andere Teile, dann folgen diese als eigene (selbstgeschriebene) Funktionen nach der Funktion main.

Zwischen {} steht die eigentliche Funktion. Als erstes wird hier die (lokale) Variable index definiert, die vom Typ int ist (Objektart int bedeutet ganze Zahlen). Lokale Variablen, das sind solche innerhalb einer Funktion, werden immer nach einer { definiert und sind bis zur dazupassenden } verwendbar. Mit einer Wertzuweisung (nur ein =) wird der Wert der Variablen index, der bisher undefiniert war, auf 0 gesetzt. Die nachfolgende while Schleife hat 2 Befehle innerhalb der Schleife. Deshalb müssen diese mit {} zu einer Einheit zusammengefaßt werden. Zu Beginn der while Schleife wird die Bedingung index < 6 geprüft. Weil diese zutrifft wird mit printf ein fixer Text samt Zeilenvorschub ausgegeben. Der Wert der Variablen index wird um 1 erhöht, die Bedingung der Schleife wird wieder geprüft usw., solange bis der Wert der Variablen index zu hoch ist.

Dieses Programm führt die Schleife sechsmal aus. Weil \n jeweils eine neue Zeile bewirkt, ergibt die Ausgabe am Bildschirm:

```
Das 1. Beispielprogramm.
Das 1. Beispielprogramm.
Das 1. Beispielprogramm.
Das 1. Beispielprogramm.
Das 1. Beispielprogramm.
Das 1. Beispielprogramm.
```

Wieviele Ausgabezeilen erzeugt das folgende Programm am Bildschirm ?

```
#include <stdio.h>
void main()
{
   printf("Das ist eine auszugebende Zeile.\n");
   printf("Das ist auch eine ");
   printf("auszugebende Zeile.\n\n");
   printf("Diese Ausgabe steht in der ?. Ausgabezeile.\n");
}
```

Der Befehl printf (genaugenommen eine Funktion und kein Befehl) kann nicht nur feste Texte (sogenannte Konstante) ausgeben, sondern auch den jeweiligen Wert von Variablen drucken. Die Wörter schreiben, ausgeben oder auch drucken bedeuten hier sinngemäß immer dasselbe. Wir definieren also im folgenden Programm eine Variable namens index und schreiben diese (gemeint ist der Inhalt) mehrmals. Das %d ist das dezimale Formatierungszeichen für die Variable index. Im printf steht als erster Parameter immer ein String (mehrere alphanumerische Zeichen) unter Anführungszeichen, der verschiedene mit % beginnende Formatierungssteuerzeichen für die als weitere Parameter folgenden Variablen enthält. Hier haben wir den Formatierstring und jeweils nur einen Parameter, die Variable index.

```
#include <stdio.h>
void main()
{
   int index;
   index = 13;
   printf("Der Wert der Variablen index ist momentan %d\n",index);
   index = 2727;
   printf("Der Wert der Variablen index ist momentan %d\n",index);
   index = 3;
   printf("Der Wert der Variablen index ist momentan %3d\n",index);
}
```

Die Programmausgabe würde folgendermaßen aussehen:

```
Der Wert der Variablen index ist momentan 13
Der Wert der Variablen index ist momentan 2727
Der Wert der Variablen index ist momentan   3
```

Ein übersichtlicher Programmierstil erleichtert das Verständnis eines Programms gewaltig. Weil Sie in C aber an keinerlei Spaltenbegrenzungen gebunden sind (der Code wird vom Compiler als ein endloser Befehlsstrom interpretiert; lediglich Preprocessoranweisungen beginnen am Zeilenanfang mit einem # und enden in derselben Zeile) können Programme auch sehr schlecht geschrieben werden.

Abschreckend, nicht wahr ?

```
#include <stdio.h>
void main() /*Das Hauptprogramm startet hier */{printf("Gute Form ");printf
             ("kann helfen ");printf("ein Programm zu verstehen.\n")
;printf("Schlechte Form ");printf("macht ein Programm ");
printf("unleserlich.\n");}
```

besser würde dieses Programm aber so aussehen

```
#include <stdio.h>
void main()  /* Das Hauptprogramm startet hier */
{
   printf("Gute Form ");
   printf          ("kann helfen ");
   printf                      ("ein Programm zu verstehen.\n");
   printf("Schlechte Form ");
   printf              ("macht ein Programm ");
   printf                                  ("unleserlich.\n");
}
```

Einrückungen sind

Wesentlich für eine saubere Programmgestaltung sind Einrückungen. Wann immer zusammengesetzte Befehle verwendet werden, bzw. wenn eine geschwungene Klammer-auf geschrieben wird sollte man die folgenden Zeilen einrücken (weiter rechts schreiben). Unter zusammengesetzte Befehle versteht man jene Struktogrammsymbole, die in ihrem Inneren weitere Anweisungen enthalten. Mit dem Ende des Befehls, bzw. bei der geschwungenen Klammer-zu wird wieder weiter links weitergeschrieben (ausgerückt).

```
{       /*   so  */                  {        /* oder auch so  */
    int variable=0;                       int variable=0;
    while (variable < 6)                  while (variable < 6) {
    {   .......;                              ........;
        variable = variable + 1;            variable = variable + 1;
    }                                     }
}                                     }
```

3.1. Beispielprogramm am Computer ausführen

Programme eines Computers sind in irgendwelchen Programmiersprachen geschrieben. Der Computer selbst (dessen CPU) versteht auch eine Art Programmiersprache (die Maschinensprache der jeweiligen CPU), diese besteht aber nur aus binären Zahlen und ist für Menschen denkbar untauglich. Deshalb wurden sogenannte problemorientierte Programmiersprachen entwickelt, die einerseits der Mensch leichter handhaben kann und die andererseits automatisch in die Maschinensprache des Computers übersetzt werden können. Unsere Sprache C wird daher von einem speziellen, Compiler genannten, Programm in jene Zahlenfolgen übersetzt, die vom Prozessor der Maschine ausgeführt werden können. Am PC sind das Maschinenbefehle aus dem Instruktionssatz des Intel 8086 Processors.

1. Mithilfe eines **Editors** (ein Programm, welches Eingabe und Speicherung von Texten unterstützt) tippen wir unser Programm am Computer und speichern es in einer Datei. Das Programm liegt nun im **Sourcecode** oder **Quelltext** vor.

2. Mit dem **Compiler** übersetzten wir das Programm in die Maschinensprache des Rechners.

3. Dabei wird unser Programm auf Syntaxfehler überprüft. Gibt es welche, dann endet die Compilierdurchführung mit Fehlermeldungen und/oder Warnungen. Wir müssen die Fehler mit Hilfe des Editors ausbessern und erneut das Compilerprogramm starten.

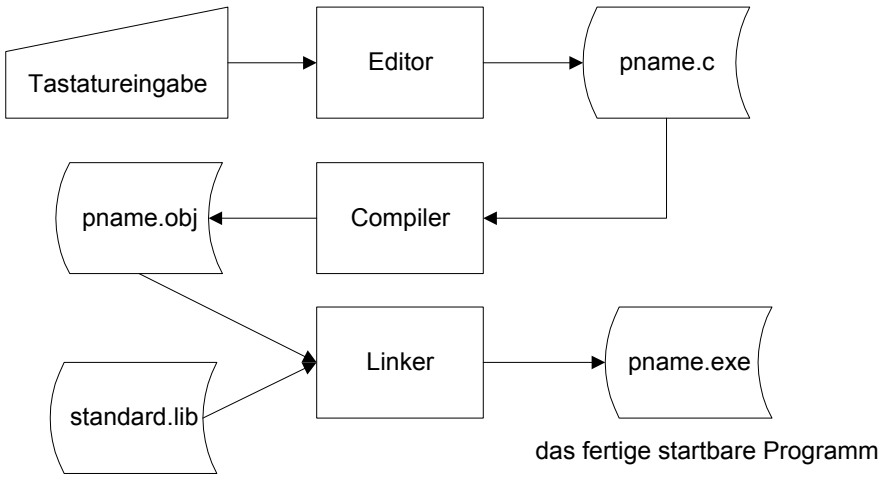
4. War unser Compiler erfolgreich, so liegt jetzt ein sogenannter **Objektmodul** vor; darin sind alle Befehle und auch Variable unseres Programms in Maschinensprache. Dieser ist aber noch unbrauchbar, weil zur Ausführung noch weitere vorgefertigte Teile bzw. weitere eigene Funktionen benötigt werden.

5. Mit dem **Linker** (Binder) werden daher alle benötigten Objektmodule zu unserem Modul dazugebunden. Jetzt ist unser Programm ein **ausführbares Programm** und kann gestartet werden.

6. Leider arbeiten Programme nicht gleich richtig und es müssen beim Testen noch semantische Fehler gefunden und korrigiert werden. Dafür kann man einen Debugger verwenden, ein Programm, welches unser Programm schrittweise ausführt und dabei auch den Inhalt von Variablen anzeigen kann. Ein Debugger ermöglicht sinngemäß einen Schreibtischtest am Computer. Wenn Fehler gefunden werden, dann muß unser Programm Quelltext geändert werden und wir beginnen wieder bei 1.

Früher wurden diese Schritte mit getrennten Programmen (Editor, Compiler, Linker, Debugger) ausgeführt. Insbesondere am PC gibt es heute Integrierte Entwicklungsumgebungen, die alle Funktionalität unter einer menügesteuerten und mausunterstützten Oberfläche anbieten. Ebenfalls mit dabei sind Hilfesysteme, die teilweise schon Zugriff und Suche auf/in allen Handbüchern der Programmiersprache ermöglichen.

Lediglich die Bedienung wurde verbessert, die Arbeitsvorgänge sind aber die gleichen geblieben, auch die jeweiligen Dateien (pname.c - Quellcode, pname.obj - Objektmodul, pname.exe - ausführbares Programm) sind noch immer die gleichen. Mit >standard.lib< liegt eine sogenannte Bibliothek (Library) vor, in der vorgefertigte Objektmodule enthalten sind.



Übungen:

- Sie müssen die Programmierumgebung Ihres Compilers in den Griff kriegen. Nehmen Sie das größere Beispielprogramm und versuchen sie es zum Laufen zu bringen. Geben Sie auch den Wert der Variable index mit aus. Versuchen Sie auch mit dem Debugger eine schrittweise Durchführung Ihres Programms.

- Setzen Sie das Struktogramm eines einfachen früheren Beispiels (aus den früheren Übungen) in ein C - Programm um und entwickeln es am Rechner..

4. Richtlinien zur vernünftigen Programmierung

Wie hinlänglich bekannt, ist die Wartbarkeit heute die wahrscheinlich wichtigste Eigenschaft eines ansonsten funktionsfähigen Programms. Daher sind die nachfolgenden beschriebenen Maßnahmen sinnvoll anzuwenden. Sollte Ihnen dieses Kapitel teilweise unverständlich vorkommen, so lesen Sie ihn am Ende des Skriptums nochmal und wenden vorläufig nur die leicht verständlichen Absätze an.

Zuerst denken, dann den Programmcode schreiben

) Schreiben Sie den eigentlichen Programmcode (ausgenommen Definitionen) erst dann, wenn Sie bis ins Detail wissen, wie er aussehen soll !!!! Wenn Sie schon früher beginnen, dann versauen die späteren Änderungen das Programm schon vor dem ersten Echtlauf. Während der Denkphase hat man die beste Gelegenheit, an der Dokumentation zu schreiben.

Modularisierung ist noch wichtiger als Strukturierung

) Zerlegen Sie das Problem sinnvoll in einzelne, jeweils möglichst abgeschlossene Teile (Module), bevor Sie daran denken, wie Sie es in der jeweiligen Sprache lösen werden. Die einzelnen Module (Unterprogramme, Funktionen) sollten nur über minimale Schnittstellen (Parameter, Funktionsrückgabewert, globale Variable) miteinander verbunden sein. Die Tätigkeit eines Moduls muß exakt in seiner Dokumentation (am besten im Programmkopf) beschrieben sein; es darf nicht notwendig sein, den Code zu lesen um den Modul verwenden zu können. Keinesfalls darf der Modul abstürzen, nur weil er mit falschen Parametern aufgerufen wird. Jeder Modul wird isoliert vom restlichen Programm vollständig ausgetestet. Kennt man die Dokumentation eines Moduls, so kann man diesen verwenden ohne seinen Code zu kennen.

C ist besonders gut für Modularisierung geeignet. Seit der Urzeit von C ist es auch üblich, daß viele kleine Module (Funktionen) ein Programm ergeben. Sie sollten daher auch ihre Beispielprogramme in getrennt compilierbare Module aufteilen.

Wichtig

NUR minimale und genau definierte Schnittstellen zwischen Modulen ermöglichen es, einen Modul isoliert zu betrachten und zu testen !!

) Schreiben Sie den Code selbst strukturiert; es sollte nie notwendig sein, zum Verständnis eines Stücks Programmcode an (vielen) anderen Programmstellen nachsehen zu müssen (Ausnahme: Definitionen, aufgerufene Module).

In C sind Befehle vorhanden, um die Elemente der Strukturierten Programmierung (Sequenz,if,while,until,case,exit) direkt umzusetzen.

) Gestalten Sie das Programm selbst lesbar. Gute Gliederung, einrücken zusammenhängender Teile, gezielte Kommentierung schwieriger Passagen sind sehr hilfreich. Es ist durchaus üblich, in C Programmen praktisch jede Zeile mit einem Kommentar zu versehen.

In C hat sich, vor allem durch den Beautifier (cb) und durch die Beispiele im Buch von Kernighan und Ritchie, ein sehr übersichtlicher Programmierstil durchgesetzt.

Die Compiler unterscheiden auch Groß- und Kleinschreibung. Das wird genutzt, um einige Teile (Symbolische Konstante und mit typedef vereinbarte Namen) vollständig groß und das restliche Programm vollständig klein zu schreiben.

) Verwenden Sie für alle Variablen sprechende Namen. Ausgenommen davon sollten nur kurzzeitig eingesetzte Workfelder bzw. Indizes sein. Wenn jemand rechenfeld statt rf schreibt, so ist das zwar ein langer aber kein sprechender Name. summe_mwst für die Summe der Mehrwertsteuerspalte wäre aber sinnvoll, vorausgesetzt die Variable wird ausschließlich dafür verwendet. Einzelne Namensteile trennt man normalerweise mit dem Underliner; beispielsweise ist "summe_gehalt_angestellte" ein brauchbarer Name. Als alternative Schreibweise wird auch "SummeGehaltAngestellte" gelegentlich angetroffen. Weil C Groß- und Kleinschreibung unterscheidet, muß man sich aber genau daran halten.

  Hier ist es mit C schlecht bestellt. Die meisten Compiler können zwar endlos lange Namen bewältigen, es ist aber leider üblich, kurze Namen zu verwenden. In diesem Punkt sollten Sie die C Tradition nicht fortsetzen.

) Die einzelnen Variablen haben Datentypen (binär, gezont). Verschiedene Operatoren können auf bestimmte Datentypen angewandt werden. Die Division ist nur mit numerischen Variablen möglich, Substring-Bildung kann nur mit gezonten alphanumerischen Variablen sinnvoll sein. Wenn die verwendeten Variablen nicht genau zusammenpassen, dann kommt es zu einer (automatischen) Konvertierung. Typenstrenge Sprachen machen keine automatischen Konvertierungen; Andere Sprachen konvertieren automatisch, was nicht immer im Sinne des Programmierers ist. Auch in C wird sehr viel automatisch konvertiert, was viel Mitdenken durch den Programmersteller erfordert.

  Verlassen Sie sich nicht (zu sehr) auf automatische Konvertierungen.

) Sehen Sie sich Ihre eigenen Programme einige Monate nach der Fertigstellung nochmals an und fällen Sie ein selbstkritisches Urteil.

) Wenn Sie noch mutiger sind, lassen Sie Ihr Programm von Kollegen begutachten, erzählen Sie aber, das Programm hätte irgendein Unbekannter verfaßt.

5. Variable, Datentypen und Konstante in C

Namen geben Sie Ihren Variablen und Funktionen

Namen von Variablen und Funktionen bestehen in C aus Buchstaben, dem Unterstreichungszeichen _ und aus Ziffern. Die erste Namensstelle darf keine Ziffer sein und sollte kein Unterstreichungszeichen sein, weil sehr viele compilerinterne Namen damit beginnen. Groß- und Kleinbuchstaben werden unterschieden !!! Es ist üblich, sogenannte symbolische Konstante (die mit dem #define definiert werden) und auch Namen aus typedef Definitionen vollständig mit Großbuchstaben, alle anderen Namen (Variable, Funktionen) vollständig in Kleinbuchstaben zu schreiben. Die Länge der Namen ist beliebig (mindestens 509 Zeichen sind möglich), jedoch werden meist nur die ersten n (31 oder mehr) Stellen vom Compiler verwendet. Es ist möglich, aber meist nicht der Fall, daß externe Namen (die der Linker kennen muß, weil der gleiche Name in mehreren Funktionen verwendet wird) auf bis zu 6 Stellen gekürzt werden. Die nebenstehenden Wörter sind reservierte Wörter und dürfen nicht als Namen verwendet werden. Verwenden Sie sprechende Namen, nützen Sie den Unterstrich zur Bildung sinnvoller Namen. Funktionsnamen (z. B. printf) sind keine reservierten Wörter. Definieren Sie trotzdem keine Variable namens printf.

Gesperrte Wörter in C :

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Variable können Inhalte in einem bestimmten Datentyp aufnehmen - gehören zu einem Objekttyp

Variable müssen vor ihrer ersten Verwendung definiert (vereinbart) werden, deshalb sollten die Variablendefinitionen am Anfang jeder Funktion bzw. jedes Programmblocks {} stehen. Je nachdem wo und wie die Variable definiert wurde, kann sie nur innerhalb des Programmblocks, innerhalb der Funktion, innerhalb des gemeinsam compilierten Programmstücks oder auch im gesamten Bereich des ausführbaren Programms (inclusive der in anderen Dateien stehenden Unterprogramme) gültig sein. Sowohl die Lebensdauer als auch der Gültigkeitsbereich von Variablen werden später noch genau besprochen.

Variablen sind im Prinzip mit folgender Syntax zu definieren:

```
[Speicherklasse] Datentyp Name[,...];       z.B.    int zahl1,zahl2;
```

Die Speicherklasse wird später erläutert; [ ] zeigt, daß sie weggelassen werden kann
Der Datentyp (hier int) kann auch aus mehreren Wörtern bestehen.
Pro Datentyp können auch mehrere Variablen definiert werden; angezeigt durch [,...].
Mit Strichpunkt werden in C alle Statements abgeschlossen.

5.1. Datentyp Ganzzahl (interne Binärzahl)

```
int a;                  /* integer ist der zum Register der Maschine   */
int wert;               /* passende Datentyp, daher 2 Byte am PC, aber */
int zahl,temp,work;     /* 4 Byte auf den meisten anderen Maschinen    */

short int de;           /* braucht man nur 2 Byte (-32768 bis 32767)   */
short int index, faktor; /* so kann man aus Platzgründen (in Sätzen von */
short var1,var2;        /* Dateien) short einsetzen                    */

long int dd;            /* braucht man 4 Byte nimmt man long und hat   */
long zaehler_input;     /* über 2 Mrd. (-2147483648 bis 2147483647)    */

unsigned short zl64;    /* braucht man keine negativen Zahlen, so ver- */
unsigned long var_gross; /* doppelt man den positiven Wertebereich mit  */
unsigned int workvar3;  /* unsigned (kein Vorzeichen)                  */
```

Wenn Sie portabel, also für mehrere Anlagen programmieren wollen, dann dürfen Sie int nur einsetzen, wenn die Länge (und damit der Wertebereich) egal ist. Weil int normalerweise dem Register der Maschine entspricht, sind auch längere int und long Typen denkbar. Mit Sicherheit gilt aber : Länge short <= Länge int <= Länge long. Die Darstellung negativer ganzzahliger Werte erfolgt intern im sogenannten 2er Komplement.

Konstanten für alle intern binären Typen dürfen keine Nachkommastellen haben, können jedoch als normale Dezimalzahlen aber auch als Oktalzahlen mit einer führenden Null und sogar als Hexadezimalzahl mit führenden 0x geschrieben werden. Numerische Konstanten (ohne Dezimalpunkt) gelten als integer Konstanten; ist der Zahlenwert zu groß, so kann durch Anfügen eines L oder l eine long integer Konstante erzeugt werden. Durch Anfügen eines u oder U erreicht man eine unsigned Konstante. Es gibt keine dezitierte short Konstante.

richtig : 123 0 -66 234573952LU 0127 +35 0xff 0xa27c 44821L 0x44L
falsch : 0678 3c61 3.14 0x6km2 12a6 0x12le12 -66u

5.2. Datentyp Gleitkomma (entspricht binärem Maschinenformat)

```
float zahl2;              /* normales Gleitkomma einfache             */
float e1,pi1;             /* Genauigkeit  (mindestens 6 Stellen)      */
double genauer;           /* doppelte Genauigkeit (mind. 10 Stellen)  */
long double noch_genauer;/* noch höhere Genauigkeit (wenn möglich)   */
```

Der Wertebereich von float - Variablen ist sehr maschinenabhängig, er reicht aber oft von 10 hoch 100 bis 10 hoch -100. Die Norm verlangt bloß einen Mindestbereich von 10^{-38} bis 10^{+38}.

Als Konstante sind die üblichen Ziffernschreibweisen mit Dezimalpunkt (kein Komma) möglich, aber auch die Exponentenschreibweise. Gleitkommakonstante gelten als double. Durch Anhängen von f,F gibt es float, durch Anhängen von l,L long double Konstante.

richtig: 3.14 2.71 1238762.45 12.73e-7 -14.3 0.12E-33 -3.2F +12E-12L
falsch: 3,14 22 2222+33

5.3. Datentyp character (nur eine Stelle)

```
char nummer;              /* der C char-Typ ist nur ein Zeichen lang  */
char a1,text;
char lastzeichen;
unsigned char  zeichen8; /* unsigned für 8 Bit langen Character      */
```

Eine character kann genau ein Zeichen aus dem Zeichensatz der Maschine aufnehmen; 7 oder 8-Bit oder auch wide (16 Bit) Code.

dezimale Codes für \Zeichen

7	\a	BEL
8	\b	BS
9	\t	HT
10	\n	LF
11	\v	VT
12	\f	FF
13	\r	CR

Konstante werden zwischen Hochkomma geschrieben. Dazwischen steht ein Zeichen; wenn dieses nicht druckbar ist, kann es auch als \c oder \nnn oder \xnn dargestellt werden. Das c steht für einen speziellen Buchstaben, nnn ist der oktale Code des jeweiligen Zeichens, bei xnn ist das nn der hexadezimale Code des jeweiligen Zeichens. Bei zukünftig möglichen Wide-Character Zeichensätzen steht ein L vor der Konstante.

\n - Neue Zeile, \t - horizontaler Tab, \v vertikaler Tab, \b - Backspace, \f Seitenvorschub, \r - Carr. return, \a - Klingel, \\ - Backslash, \0 -binäre Null, \' \" \? - immer das Zeichen selbst.

richtig: 'a' '0' '\n' '\014' '\b' '\x2f'
falsch: 0 '\j' '\876' '\o' "a" /* statt 0 soll korrekt '\0' geschrieben werden */

Zum Arbeiten mit mehreren charakters (diese werden Strings genannt = ein normaler Text) werden char - Arrays (siehe unten) verwendet. Weil es keine Operatoren gibt, die mit den char - Arrays direkt arbeiten könnten, gibt es eine Menge Funktionen und auch eine eigene String Konstante. Diese String Konstante besteht in der Regel aus mehreren Zeichen, begrenzt von Anführungszeichen.

z.B. "Das ist eine Stringkonstante" "abcd"

Das char - Array muß immer um 1 Byte größer sein als der Text in der Konstante, weil ein \0 als Endezeichen angefügt wird. Dadurch entstehen variabel lange char-Felder, die zwar einen bestimmten Speicherplatz erfordern, aber logisch nur bis zum ersten \0 im Feld reichen. Wenn Sie auf das \0 vergessen, kann das tragische Folgen haben. Wenn nämlich nach der Befüllung der Variablen durch eigene Programmteile eine Funktion aufgerufen wird, die dieses char-Array verwendet, so hat die Funktion keine Ahnung von der Feldlänge und sucht sich quer durch den Hauptspeicher bis zum nächsten \0. Die String Konstante "abcd" ist daher auch 5 Bytes lang.

Die obigen Variablen können auch problemlos bei der Definition initialisiert werden:

```
int nummer2=12;                    /* Direkt bei der Definition     */
long a2,b=33,c=12345678;           /* erhalten die Variablen ihren  */
float pi=3.14,e=2.71,eps=1.0e-5;   /* ersten Wert -> initialisieren */
char backslash='\\';
int k = 0;
```

5.4. Arrays (Zusammenfassen mehrerer Elemente gleichen Typs)

Mit dem Arraynamen können mehrere durchnumerierte Felder angesprochen werden

Es können in C ein- oder mehrdimensionale Arrays (Tabellen) definiert werden. Es werden mehrere Datenfelder (mit gleichem Datentyp) unter einem Variablennamen zusammengefaßt. Mit Hilfe des Index werden die Felder sozusagen durchnumeriert. Es ist auch möglich, die Felder nicht alle hintereinander (= eindimensional), sondern in Form eines Rechtecks (= zweidimensional) oder auch mehrdimensional anzuordnen. Der Index jeder Dimension beginnt immer von **0** weg zu laufen.

Die Arrayindizes beginnen immer mit 0 !

int tab [3]; ist ein eindimensionales Array und hat 3 Elemente (tab[0], tab[1], tab[2]). Der Index wird nach dem Tabellennamen in eckigen Klammern geschrieben und wird ab 0 durchnumeriert.

tab [0]	tab [1]	tab [2]

! Wichtig

versuchen Sie im Programm tab[3] anzusprechen, so liegen Sie außerhalb (hinter) der Tabelle und bewegen sich im Speicherbereich einer anderen Variable. Weil C (wie die meisten anderen Programmiersprachen) solche Fehler nicht erkennt, ist dies häufig die Ursache für schlimme semantische Mängel !!!

int feld [2] [3]; ist zweidimensional und definiert 6 Elemente

feld [0][0]	feld [0][1]	feld [0][2]
feld [1][0]	feld [1][1]	feld [1][2]

Dieses kurze Programmstück eignet sich zum Initialisieren (Befüllen) einer Tabelle.

```
#define TABDIM 9           /* das ist eine symbolische Konstante, die */
int tab[TABDIM],i;         /* als Dimensionsangabe verwendet wird  */
i = 0;
while (i < TABDIM)         /*  alle Elemente aus tab werden auf 0 gesetzt  */
{  tab [i] = 0;            /* alternativ: for (i = 0;i < TABDIM; i = i + 1)*/
   i = i + 1;              /*                tab [i] = 0;                  */
}
```

Die Dimensionsausdehnungen dürfen nur Konstante oder symbolische Konstante sein. Bereichsüberschreitungen (falsche Indexwerte) beim Zugriff auf das Array werden vom Compiler nicht erkannt; also Vorsicht. Was würde wohl passieren wenn Sie statt "i < TABDIM" versehentlich "i <= TABDIM" schreiben ?

Beispiele: simple Verwendung von ein- und zweidimensionalen Array.

```
#include <stdio.h>
void main()
{
int index;
int stuff[12];
float weird[12];

   for (index = 0;index < 12;index++) {
      stuff[index] = index + 10;
      weird[index] = 12.0 * (index + 7);
   }

   printf("%s\n",name1);
   printf("%s\n\n",name2);
   for (index = 0;index < 12;index++)
      printf("%5d %5d %10.3f\n",index,stuff[index],weird[index]);
}
```

```
#include <stdio.h>
#define DIM-GROSS 8
void main()
{
int i,j;
int gross[DIM-GROSS][ DIM-GROSS],maechtig[25][12];

   for (i = 0;i < DIM-GROSS;i++)
      for (j = 0;j < DIM-GROSS;j++)
         gross[i][j] = i * j;   /* enthält das Einmaleins bis 8 x 8*/

   for (i = 0;i < 25;i++)
      for (j = 0;j < 12;j++)
         maechtig[i][j] = i + j;   /* stellt eine Summentabelle dar*/

   gross[2][6] = maechtig[24][10]*22;
   gross[2][2] = 5;
   gross[gross[2][2]][gross[2][2]] = 177;/* entspricht gross[5][5] = 177;*/

   for (i = 0;i < DIM-GROSS;i++) {
      for (j = 0;j < DIM-GROSS;j++)
         printf("%5d ",gross[i][j]);
      printf("\n");               /* Newline für zeilengerechte Ausgabe */
   }
}
```

Arrays können auch direkt initialisiert werden.

```
int tab1[5]={0,1,44,2,8};
int matrix [3] [2] = { {2,3}, {1,4}, {7,5} };/* pro Index einmal {}  */
char text1[6] = {'h','a','l','l','o','\0'};  /* etwas mühsam,        */
char text2[6] = {"hallo"};        /* so gehts auch       Sonderfall nur  */
char text3[6] = "hallo";          /* am einfachsten      für char-Arrays */
```

Im normalen Programmablauf gibt es keine Möglichkeiten, Arrays als Einheit anzusprechen, so wie bei den hier gezeigten Initialisierungen. Dort muß jedes Element einzeln (z.B. tab[3]=2; text[0]='h';) verwendet werden. Bei manchen Befehlen (genauer Funktionsaufrufen) kann man aber den Arraynamen ohne [ ] angeben.

5.4.1. Spezielles zu char - Arrays

Texte als Strings (char - Arrays mit \0 am Ende !)

Wichtig

Weil der Typ char nur ein Zeichen speichern kann, ist das char-Array häufig in Verwendung um Strings zu speichern. Um mit Strings besser umgehen zu können, gibt es String Konstante (zwischen Anführungszeichen) und eine eigene Gruppe von Funktionen, die mit Strings arbeiten weil es keine Operatoren für Strings oder Arrays gibt.

Diese verlangen aber, daß ein String immer mit einer binären Null abgeschlossen wird.

Hat ein String kein \0 am Ende, so kann das, genauso wie das planlose Überschreiten von Indexgrenzen eines Arrays, zu völlig dubiosen Programmabstürzen führen.

```
#include <string.h>    /* notwendig für alle str... Funktionen */
#include <stdio.h>
void main()
{
char name1[12],name2[12],mixed[25];
char title[20];

   strcpy(name1,"Rosalinde");            /* kopiert (nach,von)           */
   strcpy(name2,"Schnecke");             /* im Zielfeld muß genug Platz  */
   strcpy(title,"Das ist der Titel.");   /* für den Text und ein \0 sein */
   printf("     %s\n\n",title);
   printf("Name 1 is %s\n",name1);
   printf("Name 2 is %s\n",name2);
   printf("Name1 ist %d Zeichen lang\n",strlen(name1));

   if(strcmp(name1,name2)>0)  /* Vergleich, liefert true if name1 > name2 */
      strcpy(mixed,name1);
   else
      strcpy(mixed,name2);

   printf("Der alphabetisch größere Name ist %s\n",mixed);

   strcpy(mixed,name1);
   strcat(mixed,"  ");
   strcat(mixed,name2);
   printf("Beide Namen lauten %s\n",mixed);
}
```

Die wichtigsten Funktionen , welche char-Array Bearbeitung unterstützen:

strlen(string);

Die Funktion **strlen()** gibt die Länge der mit ´\0´ abgeschlossen Zeichenkette **string** zurück. Das abschließende Nullbyte wird nicht mitgezählt.

strcpy(str1,str2);

Mit der Funktion **strcpy()** wird der Inhalt von **str2** nach **str1** kopiert. **str2** muß eine mit ´\0´ abgeschlossene Zeichenkette (char-Array) sein. **str1** muß lang genug sein um den Inhalt von **str2** aufzunehmen.

strcat (str1,str2);

Die Funktion **strcat()** hängt den Inhalt von **str2** an **str1** an. Das Nullbyte, das ursprünglich am Ende von **str1** stand, wird vom ersten Zeichen der Zeichenkette **str2** überschrieben. Auch hier muß **str1** lang genug sein um den Gesamtinhalt aufzunehmen.

Ergebniswert	Bedeutung
kleiner 0	str1 < str2
gleich 0	str1 == str2
größer 0	str1 > str2

strcmp(str1,str2);

Die Funktion **strcmp()** vergleicht zwei mit ´\0´ abgeschlossene Zeichenketten (gemäß Code) und gibt einen Ergebniswert zurück.

5.4.2. Suchen von Werten in Arrays

Nachdem ein Array mit Werten befüllt wurde ist es oft notwendig einen bestimmten Wert innerhalb eines Arrays zu suchen, entweder um nur seine Existenz nachzuweisen oder um seine Position (den Index) im Array festzustellen.

Das folgende Beispiel durchsucht ein Array nach Werten größer als 200 und zählt deren Vorkommen.

```
#include <stdio.h>
#define TABDIM 200

void main()
{
    int anzahl,i;
    int tabelle [TABDIM] = {22,44,66,234,55,532,345,62,91};
                           /* die restlichen Elemente werden
                              auf 0 gesetzt */
    anzahl = 0;
    for (i=0;i<TABDIM;i = i + 1)
        if (tabelle[i] > 200) anzahl = anzahl + 1;

    printf ("Es gibt %d Werte > 200 in der Tabelle\n",anzahl);
}
```

Mit einer Schleife (in diesem Fall eine for - Schleife) werden alle Elemente des Arrays sequentiell (der Reihe nach) überprüft ob sie der Bedingung größer 200 entsprechen. Wenn ja, wird das betreffende Arrayelement bearbeitet - in diesem Programm wird nur gezählt, man könnte das Element aber genausogut verändern.

Bei einem unsortiertem Array gibt es kaum sinnvollere Vorgangsweisen. Sollten die Werte innerhalb des Arrays aber in sortierter Folge vorliegen, dann müssen nicht mehr alle Arrayelemente überprüft werden. Im vorliegenden Fall würde man das Array wohl von oben nach unten durchsuchen und die Schleife verlassen, sobald man auf den ersten Wert < 201 trifft. Sucht man nur einen bestimmten Wert in einem sortierten Array, dann wird man wohl ein Verfahren anwenden, welches sich **binäres Suchen** nennt. Genauso wie jeder Mathematiker dividieren kann sollte dieser Algorithmus zum Grundwissen eines jeden Programmierers gehören.

binäres Suchen

Man sucht in einem sortierten Array die Position (den Index) eines bestimmten Werts. Selbstverständlich ist bekannt wieviele Werte im Array enthalten sind.

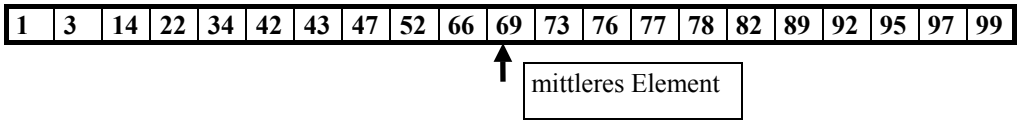
Wichtig

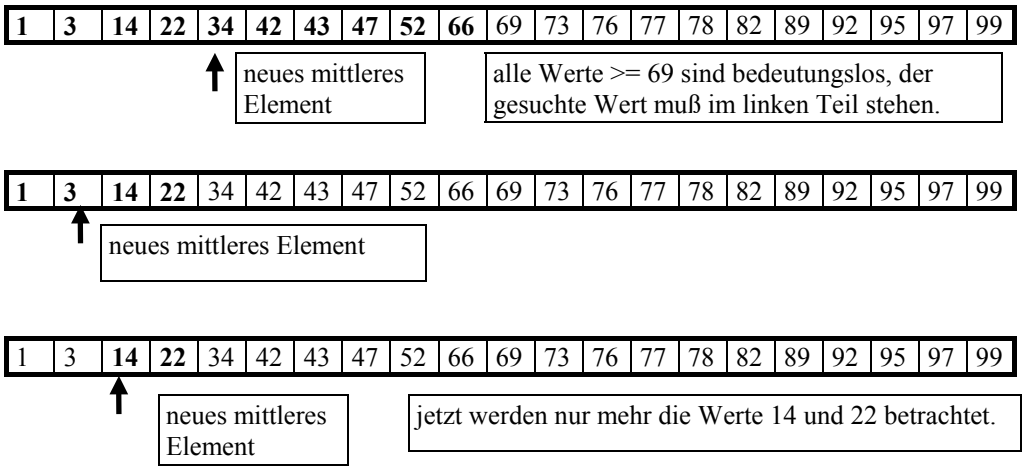
1. Errechnen des Index des mittleren Arrayelements (bei 7 Elementen das 3., bei 100 Elementen das 50., etc.)

2. Enthält dieses Element den gewünschten Wert -----> Treffer und fertig

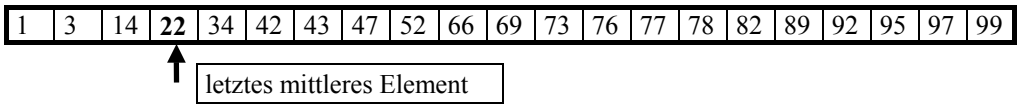
3. Ist der gesuchte Wert kleiner als der Inhalt des Elements aus Punkt 2, dann muß dieser Wert wohl im linken (unteren) Teil der Matrix stehen, ist er größer, dann im rechten Teil. Man kann also die Suche auf eine Hälfte des Arrays beschränken und errechnet nun den Index des mittleren Arrayelements innerhalb dieser Arrayhälfte und geht zu Punkt 2. Ist die Hälfte aber so klein, daß sich kein mittleres Element mehr errechnen läßt, dann gilt ------> Wert nicht gefunden, fertig.

Beispiel: Array mit 21 Elementen, sortiert. Suche Element mit dem Wert 14:




1 3 14 22 34 42 43 47 52 66 69 73 76 77 78 82 89 92 95 97 99
neues mittleres Element
alle Werte >= 69 sind bedeutungslos, der gesuchte Wert muß im linken Teil stehen.
1 3 14 22 34 42 43 47 52 66 69 73 76 77 78 82 89 92 95 97 99
neues mittleres Element
1 3 14 22 34 42 43 47 52 66 69 73 76 77 78 82 89 92 95 97 99
neues mittleres Element
jetzt werden nur mehr die Werte 14 und 22 betrachtet.

Der Wert des jetztigen „mittleren Elements“ enthält 14 -----> Treffer - gefunden.
Würde man nicht nach dem Wert 14, sondern nach dem Wert 19 suchen, dann ginge die Suche noch weiter.



Durch mehrfache Halbierung des betrachteten Arraybereichs bleibt jetzt nur mehr ein Element übrig. Dieses kann nicht mehr weiter unterteilt werden, daher endet der Algorithmus mit nicht gefunden.

Soll man diesen Algorithmus nun in Code umsetzen, so geht man meist den falschen Weg - dies ist normal, kaum jemand findet nach der Ablaufbeschreibung sogleich den günstigsten Code. Vor Ihnen haben ebenfalls kluge Leute schon Wochen aufgewendet um diesen Code zu perfektionieren.

```
#include <stdio.h>
#define TABDIM 21

void main()
{
    int untergrenze, obergrenze, i, suchwert, gefunden;
    int tabelle [TABDIM] =
              {1,3,14,22,34,42,43,47,52,66,69,73,76,77,78,82,89,92,95,97,99};
         /* werden alle Elemente initialisiert, könnte
            man auch int tabelle [] = {....} schreiben
            die Anzahl der Init-Werte bestimmen dann die Dimension */
    printf ("\nWelchen Wert wollen Sie in der Tabelle suchen ");
    scanf ("%d",&suchwert);

    untergrenze = 0; obergrenze = TABDIM - 1; gefunden = 0;
    while (gefunden == 0 && obergrenze > untergrenze)
    {
        i = (untergrenze + obergrenze) / 2; /* die Mitte berechnen */
        if (tabelle [i] == suchwert)
            gefunden = 1; /* bewirkt Ende der while - Schleife */
        else if (tabelle [i] > suchwert)
                 obergrenze = i - 1;
             else untergrenze = i + 1;
    }
    if (gefunden == 1)
         printf ("Gefunden, der Wert steht im Element mit Index %d\n",i);
    else
         printf ("Wert nicht in der Tabelle enthalten\n");
}
```

5.4.3. Sortieren von Werten in Arrays

Man kann Arrays sequentiell (der Reihe) nach befüllen und dann extra sortieren, ebenso kann man beim Einfügen eines neuen Elements bereits die richtige Position suchen und das Element dort einfügen. Dabei muß man alle hinteren (rechten) Elemente um eine Position verschieben. Man kann auch Dateien sortieren, die auf Platten gespeichert sind. Im praktischen Einsatz wird heute die meiste Sortierarbeit im Rahmen von Datenbanksprachen erledigt.

Die Wissenschaft (=Universität) und daher auch unser Programmierunterricht versteht unter Sortieren normalerweise ein gefülltes Array, auf welches man einen Sortieralgorithmus anwendet, der alle Elemente in aufsteigende oder absteigende Reihenfolge bringt. Auch dies ist ein Gebiet, wo kluge Köpfe schon vor Ihrer Zeit taugliche (man könnte auch sagen perfekte) Algorithmen ausgefeilt haben, die Sie am besten verwenden und wenn möglich verstehen sollten. Hier folgen aber nur triviale Sortieralgorithmen, wir sind ja auch erst im ersten Drittel des Skriptums.

Selection Sort

(Select Sort, Sortieren durch direktes Auswählen, Sortieren durch Minimumsuche, Methode des "kleinsten Elementes")

Man sucht das kleinste Element im Array und tauscht es gegen das an erster Stelle befindliche aus, anschließend sucht man das zweitkleinste Element (wird dadurch gelöst, daß man im Array ab der zweiten Stelle das kleinste sucht) und tauscht es gegen das an zweiter Stelle befindliche aus usw.
Der Positionszeiger rückt immer weiter, links von ihm ist bereits alles sortiert; nur mehr der Rest wird durchsucht.

Man benötigt ungefähr $n^2/2$ Vergleiche und n Austauschoperationen.

Bubblesort

(Sortieren durch direktes Austauschen, "Sprudelmethode")

Elementares Sortierverfahren, auch dieses muß jedermann/frau im Schlaf beherrschen!

Benachbarte Elemente, beginnend mit dem ersten, werden verglichen und, wenn nötig, gemäß der Sortierreihenfolge ausgetauscht.
Der Vergleich a[i] > a[i+1] (wenn TRUE wird getauscht) führt dazu, daß nach dem ersten Durchlauf das größte Element an der letzten Stelle steht (nach n-1 Vergleichen).

Ungünstigster Fall: Wenn der kleinste Wert an der letzten Stelle steht (Turtle), dann müssen n-1 Durchläufe gemacht werden, bis er an die erste Stelle kommt.

(n-1) Vergleiche * (n-1) Durchläufe = $(n-1)^2$ Vergleiche, das entspricht etwa n^2. Im günstigsten Fall eines bereits sortierten Arrays aber nur n-1 Vergleiche.

1. Verbesserung: das Array abwechselnd in beide Richtungen durchlaufen (Shakersort) - der vorher ungünstigste Fall wird damit sehr effektiv behandelt

2. Verbesserung: zu Beginn nicht das n-te mit dem (n-1)-ten Element vergleichen, sondern den Vergleich zuerst in größeren Abständen (n/2) durchführen und den Vergleichsabstand laufend halbieren bis er 1 wird (Shell Sort) - die Laufzeit wird dabei wesentlich verbessert..

Als Beispiel folgt hier nur die einfache Variante des Bubblesort.

```
#include <stdio.h>   /* Bubble Sort */
#define TABDIM 21
#define FALSE 0      /* die üblichen defines für die Wahrheitswerte */
#define TRUE 1

void main()
{
    int i, sortiert, tausch;
    int tabelle [TABDIM] =
{77,97,14,47,22,34,95,42,99,1,43,69,73,3,76,78,82,52,89,92,66};

    printf ("\nTab unsortiert : ");
    for (i=0;i<TABDIM;i++) printf ("%d ",tabelle[i]);

    sortiert = FALSE;
    while (sortiert == FALSE)
    {
        sortiert = TRUE;
        for (i=0;i<TABDIM-1;i++)
            if (tabelle [i] > tabelle[i+1])  /* Vergleich der Nachbarn */
            {
                tausch = tabelle [i];        /* wenn nötig Tausch */
                tabelle[i] = tabelle [i+1];
                tabelle[i+1] = tausch;
                sortiert = FALSE;
            }
    }

    printf ("\nTab   sortiert : ");
    for (i=0;i<TABDIM;i++) printf ("%d ",tabelle[i]);
}
```

Sie sehen hier die Änderung der Reihenfolge nach jedem Durchlauf der inneren for - Schleife. Während die größeren Werte (97, 99) schon nach wenigen Durchläufen rechts angelangt sind werden kleine Werte (1, 3) pro Durchlauf immer nur um ein Position nach links bewegt.

77	97	14	47	22	34	95	42	99	1	43	69	73	3	76	78	82	52	89	92	66
77	14	47	22	34	95	42	97	1	43	69	73	3	76	78	82	52	89	92	66	99
14	47	22	34	77	42	95	1	43	69	73	3	76	78	82	52	89	92	66	97	99
14	22	34	47	42	77	1	43	69	73	3	76	78	82	52	89	92	66	95	97	99
14	22	34	42	47	1	43	69	73	3	76	77	78	52	82	89	66	92	95	97	99
14	22	34	42	1	43	47	69	3	73	76	77	52	78	82	66	89	92	95	97	99
14	22	34	1	42	43	47	3	69	73	76	52	77	78	66	82	89	92	95	97	99
14	22	1	34	42	43	3	47	69	73	52	76	77	66	78	82	89	92	95	97	99
14	1	22	34	42	3	43	47	69	52	73	76	66	77	78	82	89	92	95	97	99
1	14	22	34	3	42	43	47	52	69	73	66	76	77	78	82	89	92	95	97	99
1	14	22	3	34	42	43	47	52	69	66	73	76	77	78	82	89	92	95	97	99
1	14	3	22	34	42	43	47	52	66	69	73	76	77	78	82	89	92	95	97	99
1	3	14	22	34	42	43	47	52	66	69	73	76	77	78	82	89	92	95	97	99
1	3	14	22	34	42	43	47	52	66	69	73	76	77	78	82	89	92	95	97	99

Einfügesort

Direkt beim Einfügen der Daten ins vorher leere Array wird sortiert. Die neue Zahl wird mit der im Array an letzter Stelle stehenden (bereits sortierten) Zahl verglichen.
Ist diese größer als die neue Zahl, so wird sie im Array um eine Stelle "nach rechts" gerückt und der Vergleich wird mit der vorletzten Zahl im Array wiederholt.

Anschließend wird weiter so verfahren, bis die letzte bereits sortierte Zahl kleiner als die neue Zahl ist. Rechts von dieser wird die neue Zahl an die freie Stelle eingefügt.

Beim Microsoft Compiler ist eine Sortdemo enthalten, die sehr anschaulich alle Sortierverfahren vorführt (auch Quicksort, etc.).

In den C - Bibliotheken (Codestücke, die von klugen Leuten schon früher geschrieben wurden und mit dem Compiler mitgeliefert werden) finden sich vordefinierte, verwendbare Funktionen für das binäre Suchen und für den Quicksort (wesentlich besser als die hier vorgestellten Verfahren). Allerdings erfodert deren Einsatz noch weitere C Kenntnisse, am Ende des Skriptums finden sie aber entsprechende Hinweise, auch können Sie die Hilfe des Compilers studieren (F1 oder STRG-F1).

Übungen:

- Ermitteln Sie die Summe und den Durchschnitt aller Werte eines numerischen Arrays.
- Entfernen Sie aus einem numerischen Array alle negativen Werte, indem Sie die rechts stehenden Elemente nach links schieben und ganz rechts mit Nullen auffüllen.
- Ein char-Array (abgeschlossen mit \0) enthält nur Ziffern. Ermitteln Sie wie oft jede Ziffer vorkommt.
- Ein char-Array (abgeschlossen mit \0) enthält beliebige Zeichen. Ermitteln Sie wie oft jedes einzelne Zeichen vorkommt. In der Ausgabe der Häufigkeitsverteilung sollen nur Zeichen aufscheinen, die mindestens einmal im Array vorkommen. Hinweis: Zeichen werden durch Code (Ascii) dargestellt.
- Ermitteln Sie das erste und das letzte Vorkommen der Zahl 3 in einem Array.
- Sortieren Sie ein Array unter Verwendung des Selection Sort.
- Verbessern Sie den Bubble Sort, indem Sie eine der vorgeschlagenen Erweiterungen einbauen
- Schreiben Sie ein Programm, welches das Einmaleins in optisch guter Darstellung am Bildschirm ausgibt.

5.5. Beispiele für Ein- Ausgabe verschiedener Datentypen

Damit man bequem Ein- und Ausgabebefehle (= Input/Output = I/O) durchführen kann, gibt es in C die sogenannte Standardeingabe (normal über Tastatur) und Standardausgabe (normal am Bildschirm), die immer verfügbar ist. Später im Skriptum wird auch Ein- Ausgabe auf Dateien (Daten auf einer Diskette oder Festplatte) verwendet.

Weil die Ein- Ausgabebefehle je nach Objektart (Datentyp) der Variablen unterschiedlich sind, werden an dieser Stelle die verschiedenen Möglichkeiten gezeigt. Alle Befehle sind auf allen C-Compilern aller Computer gleich (entsprechen der C-Norm) und werden auch mit Dateien in ähnlicher Form funktionieren. Wenn man sie verwendet, dann muß man #include <stdio.h> in den Programmkopf schreiben. Statt Variablenname setzt man jeweils den Namen der eigenen Variablen ein.

		Eingabebefehle	Ausgabebefehle
int	Ganzzahl	scanf("%d",&Variablenname);	printf("%d",Variablenname);
float	Gleitkomma	scanf("%f",&Variablenname);	printf("%f",Variablenname);
char	ein Zeichen	Variablenname = getchar(); scanf("%c",&Variablenname);	printf("%c",Variablenname); putchar(Variablenname);
char[80]	mehrere Zeichen	fgets(Variablenname,sizeof(Variablenname),stdin) gets (Variablenname); scanf("%s",&Variablenname);	printf("%s",Variablenname); puts(Variablenname);

Wenn es mehrere Alternativen gibt, dann ist die jeweils erste Zeile zu empfehlen. Bei der Ausgabe sieht man, daß es vollkommen ausreichend ist, wenn man printf gut beherrscht, deshalb folgt etwas später eine genaue Erläuterung von printf.
Bei der Eingabe ist scanf für Zahlen aller Art die erste Wahl, für einzelne Zeichen kann man sowohl scanf als auch getchar verwenden, will man eine ganze Zeile einlesen (alles was der Benutzer tippt bis er die return Taste betätigt), dann ist fgets (oder gets) notwendig. scanf kann auch mehrere Zeichen einlesen, tut dies aber nur bis zum nächsten Leerzeichen.

lese Zahlen
bis Eingabe 0;
schreibe summe

Zahlensumme
Variable : zahl, summe
summe = 0
lese erste zahl
while (zahl ungleich 0)
summe = summe + zahl
lese zahl
schreibe summe

```
#include <stdio.h>
void main ()
{
   int summe, zahl;

   summe = 0;
        /* Leseschleife mit scanf (lesen einzelne Zahlen) */
   scanf("%d",&zahl);         /* zeichen muß int sein, sonst */
   while (zahl != 0)
   {       summe = summe + zahl;
           scanf("%d",&zahl);
   }
   printf ("Die Summe aller Zahlen ist %d\n",summe);
}
```

Bei diesem Beispiel ist es noch notwendig als Abschluß aller eingegebenen Zahlen eine 0 zu tippen, damit die Bedingung der while - Schleife unwahr wird und das Programm somit nach der Schleife weitermacht. Alle Eingabebefehle bieten

aber auch die Möglichkeit, das Ende einer Eingabe selbst zu erkennen, weil das später beim Lesen aus Dateien unbedingt nötig ist. Will man auf der Tastatur ein Ende der Eingabe simulieren, dann muß man Ctrl-Z (Strg-Z auf deutschen Tastaturen) tippen. Dieser letzte Tastendruck signalisiert EOF (End of File = Ende der Datei = Ende Eingabe). Der jeweilige Lesebefehl liest dann keine Werte in die Variable, sondern signalisiert nur Ende.

Will man bei den verschiedenen Lesebefehlen das Ende der Eingabe feststellen, dann muß man folgendermaßen vorgehen:

```
int  zahl, rueck; /* für scanf */
int  zeichen;     /* für getchar verwendet, obwohl zahl       */
char c;           /* character Typ                          */
char text [80];   /* String (char-array)                    */

rueck=scanf("%d",&zahl);
      /* liest Zahlenwerte ein und stellt sie in die Variable zahl.
         Der Wert der Variablen rueck zeigt nachher ob das Einlesen
         erfolgreich war.
         rueck = 1  -> das Einlesen war erfolgreich, in der Variablen'
                       zahl steht der gelesenen Wert.
         rueck = 0  -> es wurden keine Ziffern, sondern andere Zeichen
                       eingegeben. Die Variable zahl wurde nicht ver-
                       ändert und hat noch den früheren Wert.
         rueck = -1 -> Ende der Eingabe. Die Variable zahl wurde nicht
                       verändert und hat noch den früheren Wert.
                       Weitere Lesebefehle sind sinnlos, die
                       Leseschleife muß beendet werden !

zeichen = getchar ();
      /* getchar liest genau ein Zeichen aus der Standardeingabe
         (normal der Tastatur zugeordnet). Jedes denkbare Zeichen
         (es gibt gemäß Ascii Code 256 Zeichen) kann gelesen werden.
         Deshalb ist die Variable zeichen vom Typ int.

         zeichen zwischen 0 und 255  -> ein Zeichen wurde eingelesen.
         zeichen = -1                -> Ende der Eingabe erreicht.

            algemein gilt : schreibt man char-Variable = int-Variable
                            so wird der int-Zahlenwert als Ascii-Code
                            des Zeichens verwendet.
                            schreibt man int-Variable = char-Variable
                            so wird der Ascii Code des Zeichens als
                            Zahlenwert in die int-Variable gestellt.

rueck = fgets(text,sizeof(text),stdin);
      /* liest eine Zeile (mehrer Zeichen mit der return-Taste am Ende)
         aus der Eingabe und hängt hinten ein '\0' an.
         Dadurch entsteht ein korrekter String (mehrere Zeichen in
         einem char-Array mit '\0' als Abschlußzeichen) im Array text.
         Ist die gelesene Zeile länger als 79 Zeichen, dann wird sie
         aufgeteilt und mit mehreren Lesebefehlen eingelesen. Am Ende
         einer fertig gelesenen Zeile steht die return Taste ('\n').
         rueck != 0  -> Zeile wurde gelesen
         rueck = 0   -> Ende der Eingabe

         rueck = gets(text) würde ähnlich funktionieren, allerdings
         darf die maximale Eingabezeilenlänge niemals 79 (Länge text -1)
         übersteigen, weil das Programm sonst abstürzen oder falsch
         weiterlaufen kann.  */
```

Die folgenden Programme zeigen daher taugliche Leseschleifen für verschiedene Fälle. Zuerst nochmal die Summierung der eingegebenen Zahlen. Beachten Sie, daß nur bei korrekt gelesenen Zahlen addiert wird, wenn Fehler auftreten, so würden die falschen Zeichen weiter in der Eingabe verbleiben, deshalb werden sie mit fflush (stdin) entfernt.

Zahlensumme

Zahlensumme		
Variable : zahl, summe		
summe = 0		
lese erste zahl		
while (nicht EOF)		
if (Zahl gelesen)		
then		else
summe = summe + zahl		lösche Fehler Zeichen
lese nächste zahl		
schreibe summe		

```
#include <stdio.h>
void main ()                    /* wieder das Summierungsbeispiel */
{
   int summe, zahl, rueck;

   summe = 0;
   rueck = scanf("%d",&zahl);
   while (rueck != -1)
   {       if (rueck == 1)                 /* nur verarbeiten, wenn OK */
                   summe = summe + zahl;
           else    fflush(stdin);          /* falsche Eingaben löschen */

           rueck = scanf("%d",&zahl);
   }
   printf ("Die Summe aller Zahlen ist %d\n",summe);
}
```

Als minimales Kopierprogramm kann man das folgende Beispiel betrachten. Jeweils ein Zeichen wird eingelesen und wieder geschrieben.

Kopierprogramm

Kopieren	
Variable : zeichen, chr	
lese erstes zeichen	
while (nicht EOF)	
	schreibe zeichen
	lese zeichen

```
#include <stdio.h>  /* hier enthalten ist ein  #define  EOF -1  */
void main ()        /* dies ist ein Kopierprogramm, alle eingelesenen   */
{                   /* Zeichen kommen unverändert in die Ausgabe        */
   int zeichen;
   char chr;

   zeichen = getchar();                 /* zeichen muß int sein, sonst */
   while (zeichen != EOF)               /* kann EOF nicht erkannt werden */
   {       chr = zeichen;
           printf ("%c",chr);
           zeichen = getchar();
   }
}
```

Im Gegensatz zu anderen Sprachen, wo es für das Einlesen Sprachbefehle gibt, sind in C alle Lese- und Schreibaktivitäten durch Funktionen gelöst. Das wiederum ermöglicht, daß direkt in der Bedingung einer while-Schleife nicht nur die EOF Abfrage steht, sondern auch die Lesefunktion geschrieben wird. Dies ist mit allen drei Lesefunktionen möglich.

```
#include <stdio.h>  /* ebenso möglich ist daher diese Variante          */
void main ()        /* dies ist ein Kopierprogramm, alle eingelesenen   */
{                   /* Zeichen kommen unverändert in die Ausgabe        */
   int zeichen;
   char chr;

   while ((zeichen = getchar() ) != EOF)
   {       chr = zeichen;
           printf ("%c",chr);
   }
}
```

Beispiele für die Ausgabe von Variablenwerten.

Verschiedene Ausprägungen des printf Befehls (genauer der Funktion printf) :

Wichtig

```
#include <stdlib.h>    /* Diese beiden #include sollten in   */
#include <stdio.h>     /* jedem Programm stehen              */
main()  /* es wird das Ausdrucken von Variablen demonstriert */
{  int a;              /* gewöhnliche integer Typ            */
   long int b;         /* long integer Typ                   */
   short int c;        /* short integer Typ                  */
   unsigned int d;     /* unsigned integer Typ               */
   char e;             /* character Typ                      */
   float f;            /* Gleitkomma Typ                     */
   double g;           /* Gleitkomma, doppelte Genauigkeit   */
   char st[6]="jaja";  /* String (char-array)                */
```

```
   a = 1023;              /* irgenwelche Werte in die Variablen */
   b = 2222;
   c = 123;
   d = 1234;
   e = 'X';
   f = 3.14159F;
   g = 3.1415926535898;

   printf("a = %d\n",a);       /* dezimale Ausgabe      a = 1023      */
   printf("a = %o\n",a);       /* oktale Ausgabe        a = 1777      */
   printf("a = %x\n",a);       /* hexadezimaler Output  a = 3ff       */
   printf("b = %ld\n",b);      /* dezimal, long Var.    b = 2222      */
   printf("c = %hd\n",c);      /* dezimal, short Var.   c = 123       */
   printf("d = %u\n",d);       /* unsigned Variable     d = 1234      */
   printf("e = %c\n",e);       /* character             e = X         */
   putchar(e); putchar ('\n');/* schreibt einen char   X             */
   printf("f = %f\n",f);       /* floating output       f = 3.141590  */
   printf("g = %f\n",g);       /* double gerundet (%lf) g = 3.141593  */
   printf("a = %d\n",a);       /* einfacher dec. output a = 1023      */
   printf("a = %7d\n",a);      /* output auf 7 Stellen  a =    1023   */
   printf("a = %07d\n",a);     /* 7 Stellen mit Nullen  a = 0001023   */
   printf("a = %+7d\n",a);     /* immer mit Vorzeichen  a =   +1023   */
   printf("a = %-7d\n",a);     /* output linksbündig    a = 1023      */
   printf("f = %f\n",f);       /* normaler float output f = 3.141590     */
   printf("f = %12f\n",f);     /* output 12-stellig     f =     3.141590 */
   printf("f = %12.3f\n",f);   /* 3 Nachkommastellen    f =        3.142 */
   printf("f = %12.5f\n",f);   /* 5 Nachkommastellen    f =      3.14159 */
   printf("f = %-12.5f\n",f); /* wieder linksbündig    f = 3.14159      */
   printf("String=%s\n",st);   /* character arrays      String=jaja      */
   puts (st);                  /* nur 1 char-array      jaja             */
}
```

Die Funktion putchar (char) schreibt genau einen Character auf den Bildschirm (genauer in die Standardausgabe auf die momentane Position). Der in Klammer stehende Parameter muß also eine character Variable oder Konstante sein.

Die Funktion puts (char[]) schreibt den Inhalt eines mit \0 abgeschlossenen Strings in die Standardausgabe und hängt noch ein \n (Cursor steht somit am Anfang der nächsten Zeile) an. Als Parameter wird der Name eines char-Arrays (in dem sich eine binäre Null befinden muß) oder eine Stringkonstante verwendet.

Die Funktion printf ist die formatierte Ausgabemöglichkeit; der erste Parameter ist eine String Konstante, in der % Formatierelemente enthalten sein können. Je nach Anzahl und Art der Formatierelemente gibt es noch weitere Parameter, deren Datentypen zu den % Formatierelementen passen müssen. In der Regel gibt es für jedes %format im ersten Parameter jeweils einen weiteren Parameter. Die %Formatierstrings sind folgendermaßen aufgebaut:

Erklärung der printf Ausgabeformatierung

% [Steuerzeichen...][min-Feldbreite][**.**Genauigkeit][Längenangabe]Format

Wichtig

Steuerzeichen:	0	numerische Werte mit führenden Nullen
	-	linksbündige Ausrichtung (bei numerischen Formaten)
	+	bei Zahlen Vorzeichen immer ausgeben
	blank	immer Platz für Vorzeichen (+ = blank, - = -)
min-Feldbreite		minimale Feldbreite, die Ausgabe kann breiter, aber nicht schmäler sein, je nach Format links- oder rechtsbündig
.Genauigkeit		bei Strings (%s) die maximal auszugebende Zeichenanzahl, bei Gleitkomma die Anzahl Nachkommastellen
Längenangabe		h für short, l oder L für long Zusätze
Format		d, i, o, x, u - unterschiedliche int Varianten; c - einzelne Character, s - Strings (char-Arrays) mit \0 am Ende; f - Gleitkomma; e - Gleitkomma in Exponentendarstellung; g - ist wertabhängig ein f oder e

Feldbreite und Genauigkeit können auch erst beim Programmablauf festgelegt werden. Im Formatierelement muß jeweils * angegeben werden, in der Parameterliste muß für jeden * ein Parameter vom Typ int vorhanden sein, der den jeweiligen Wert enthält: printf ("text = %*.*s\n", ar_feldbreite, ar_genauigkeit,"*********");

5.6. Spezielle I/O für Tastatur, Bildschirm

Die bisher geschilderten I/O Funktionen sind für die Standardeingabe bzw. Ausgabe ausgelegt. Weil man diese mit <, > auch umleiten kann, müssen sie also auch mit Dateien funktionieren. Daher bieten die I/O Funktionen keine speziellen Features für Tastatur und Bildschirm. Auch verwenden sie Pufferung. Die Tastatur unterliegt normalerweise einer zeilenweisen Pufferung. Sie müssen eine ganze Zeile eingeben und mit return abschließen, bevor der Puffer an Ihr Programm übergeben wird. Dies ist bei Kommandozeileneingabe auch durchaus sinnvoll, denn so kann man unabhängig vom Programm mit der Backspacetaste auch mal falsche Eingaben korrigieren. Es wird eben ein Tastendruck nicht sofort, sondern erst nach der Return-Taste an das Programm übergeben. Bei fgets spielt das keine Rolle, bei der getchar Schleife müssen Sie aber bedenken, daß erst nach dem Drücken der return Taste alle Zeichen gelesen werden und auch das in der Schleife befindliche printf erst dann aktiv wird. Soll Ihr Programm auf jeden Tastendruck sofort und unverzüglich reagieren können, dann finden Sie mit den Funktionen aus der <stdio.h> nicht das Auslangen.

<conio.h> bietet neue Möglichkeiten, bringt aber Abhängigkeit von bestimmten Compilern.

Die Programme sind nicht mehr portabel.

Sie müssen Funktionen aus der <conio.h> verwenden, die aber nicht genormt sind und nur auf den PC Compilern funktionieren.

Für die Eingabe kann man verwenden:

getche() wie getchar(), jedes getippte Zeichen wird aber sofort ans Programm geliefert. Tippt der Anwender aber auf gar keine Taste, dann wartet getche natürlich auch bis die nächste Taste gedrückt wurde.

getch() wie getche(), das getippte Zeichen ist aber nicht wie sonst üblich am Bildschirm zu sehen. Normalerweise ist bei der Tastatur immer das sogenannte „Echo“ aktiv, welches getippte Zeichen auch am Schirm zeigt. Bei getch nicht.

Während getchar() nur Buchstaben, Ziffern und Sonderzeichen lesen kann ist es mit getch/getche auch möglich das Drücken von Funktionstasten (1-10), Cursortasten und der 6 Tasten oberhalb der Cursortasten zu registrieren. In diesen speziellen Fällen werden quasi zwei Zeichen geliefert. Beim erstenmal liefert getch() den Rückgabewert 0; man muß nochmal mit getch() lesen und erhält nun einen Code für die jeweilige Taste. Schreiben Sie selbst ein kleines Versuchprogramm um die Werte zu testen.

Für die Ausgabe kann man verwenden (Achtung ! nur beim Borland Compiler) :

clrscr() löscht den Bildschirm

gotoxy(x-Wert, y-Wert) - normalerweise wird eine Ausgabe am Bildschirm genau dorthin geschrieben, wo gerade der Cursor steht. Will man den Cursor versetzen, so kann man gotoxy verwenden. Beachten Sie, daß x die Spalte und y die Zeile ist.

Während getch und getche noch auf allen PC - Compilern funktioniern, sind diese Ausgabebeispiele nur mehr bei einem bestimmten Compilerhersteller verfügbar. Im allgemeinen sollte man versuchen mit C-Funktionen auszukommen, die genormt und überall verfügbar sind. Beispielsweise würden die beiden printf-Befehle genau das gleiche tun, wenn man den Treiber ansi.sys in der config.sys geladen hat.

```
printf("\033[2J")                    /* Bildschirm löschen */
printf("\033[%d;%dH",zeile,spalte)   /* Cursor positionieren */
```

Übungen:

- Die Anzahl der Ziffern einer positiven ganzen Zahl ermitteln, die in einen long Datentyp eingelesen wird

- Text einlesen und die einzelnen Zeilen verkehrt ausgeben

- Entwickeln Sie ein Programm, welches Körpergrößen von Schülern in cm einliest. Nachdem alle Werte eingetippt wurden soll eine Häufigkeitsverteilung ausgegeben werden.
  z.B.: 150 cm : 2 Schüler
  151 cm : 4 Schüler
  152 cm : 0 Schüler ..... und so weiter für den Bereich von 50 - 250 cm.

- Es soll ein beliebiger Text eingelesen werden. Zählen Sie, wie viele Doppelbuchstaben (wie hier pp) sich im Text befinden.

Obwohl es an dieser Stelle etwas unmotiviert erscheint sei Ihnen dieses Beispiel nicht vorenthalten. Es zeigt sogenannte Typkonvertierungen in C. Man kann Variable verschiedenen Datentyps (gilt nicht bei Arrays) einander zuweisen. Dabei wird der zugewiesenen Wert aber manchmal verstümmelt bzw. verändert.

```
void main()
{  int a,b,c;           /* -32768 bis 32767 möglich bei Pcs  */
   unsigned char x,y,z; /* 0 bis 255 (oder 127) keine negativen Werte */
                        /* char kann auch als 1-Byte unsigned int angesehen
                           werden; der Code des Zeichens ist der Wert */
   float num,toy,thing; /* Gleitkomma mit einfacher Genauigkeit */

   a = b = c = -27;     /* allen 3 Variablen (a,b,c) wird -27 zugewiesen */
   x = y = z = 'A';
   num = toy = thing = 3.6792;

   a = y;              /* a ist nun 65 (character A) */
   x = b;              /* x hat nun irgendeinen undefinierten Inhalt */
   num = b;            /* num ist jetzt  -27.00 */
   a = toy;            /* a enthält jetzt 3 */
}
```

6. Befehle in C

6.1. Das leere Statement

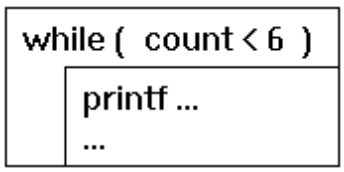
; ein einsamer Strichpunkt, der normalerweise jedes Statement abschließt, ist die Leeranweisung. Benötigt wird diese leere Anweisung beispielsweise beim if Statement; z.B. if (...) /* then */ ; else statement;

6.2. Zuweisung

Variablen wird ein Wert oder das Ergebnis eines Ausdrucks zugewiesen. Die Zuweisung ist in C eigentlich kein Befehl, weil sie nur aus dem Operator = und seinen Operanden besteht. Links vom = muß ein Ausdruck stehen, der über einen definierten Speicherplatz verfügt (z.B. Variablennamen, indizierte Arrayelemente, Ausdrücke mit dem * Operator).

```
void main ()   /* Dieses Programm soll die normale Zuweisung zeigen  */
{
  int a,b,c;    /* Integer Variablen für dieses Beispiel */

   a = 12;
   b = 3;
   c = a + b;          /* simple Addition */
   c = a - b;          /* simple Subtraktion */
   c = a * b;          /* simple Multiplikation */
   c = a / b;          /* simple Division */
   c = a % b;          /* Rest der Division a/b (Modulo, remainder) */
   c = 12*a + b/2 - a*b*2/(a*c + b*2); /* in welcher Reihenfolge werden */
   c = c/4+13*(a + b)/3 - a*b + 2*a*a; /* die Operatoren abgearbeitet ? */
   a = a + 1;          /* Inkrement von a */
   b = b * 5;
   a = b = c = 20;     /* mehrfache Zuweisung a,b,c werden auf 20 gesetzt */
   a = b = c = 12*13/4;
}
```

6.3. while - Schleife



Die Bedingung wird vor dem Schleifendurchlauf geprüft!

while (Bedingung) Aktion

Wichtig

Die Bedingung wird überprüft - wenn sie TRUE ist,
dann wird die Schleife durchlaufen (die Aktion ausgeführt), danach wird wieder die Bedingung geprüft usw. solange bis die Bedingung FALSE ergibt, dann ist die Schleife zu Ende.
Die Bedingung steht immer in (). Besteht die Aktion aus mehreren Statements, dann sind diese in {} einzuschließen (Programmblock).

```
#include <stdio.h>
void main() /* wie man sieht ein while Beispiel */
{
int count;
   count = 0;
   while (count < 6)
   {   printf("Der Wert von count ist %d\n",count);
       count = count + 1;
   }                              /* Durchläufe: 0, 1, 2, 3, 4, 5 */
}                                 /* count hat nachher den Wert 6 */
```

codieren Sie keine **Endlosschleifen**

Wichtig

Die while Schleife ist in Programmen immer anzuwenden, wenn bestimmte Anweisungen wiederholt ausgeführt werden müssen. Beachten Sie aber, daß mindestens ein Befehl innerhalb der Schleife dazu führen muß, daß die Bedingung irgendwann false wird, weil ansonsten die Schleife ewig läuft und das Programm nie mehr zu seinem geplantem Ende kommt.

Hier folgt eines der vielen Programme, das Schleifen benötigt. Die äußere Schleife sorgt für die Eingaben durch den Benutzer (!erstmals in diesem Skriptum erhält der Benutzer eine Eingabeaufforderung - was halten Sie davon) und das Programmende. Die innere (geschachtelte) Schleife führt die Potenzierungsberechnung durch.

```
#include <stdio.h>
void main ()                  /* Programm berechnet  x hoch y */
{  int x=0,y=0, yausgabe;
   double ergebnis=0;

   printf ("\nProgramm zur Berechnung von x hoch y für Zahlen > 0\n\n");

   printf ("x hoch y : Bitte X, Leerzeichen und Y eingeben; return Taste ");
   scanf("%d",&x);
   scanf("%d",&y);
   while (x > 0 && y > 0)   /* && bedeutet  und */
   {
       ergebnis = x;
       yausgabe = y;        /* wozu gibt es die Variable yausgabe ?? */
       while (y > 1)
       {
           ergebnis = ergebnis * x;
           y = y - 1;
       }
       printf("%d hoch %d ====> %.0f \n\n",x,yausgabe,ergebnis);

       printf ("x hoch y : Bitte X, Leerzeichen und Y eingeben\n");
       printf ("return Taste drücken. Ende mit Nullen          ");
       x = y = 0;          /* damit wird Schleifenende erreicht, */
       scanf("%d",&x);     /* auch wenn scanf nichts einliest    */
       scanf("%d",&y);
   }
   printf ("Bye\n\n");
}
```

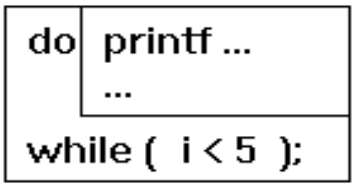
Ein beliebtes Beispiel ist das Zählen von Worten, Zeilen, Zeichen in einem beliebigen eingegebenem Text.

```
#include <stdio.h>     /* zählen von Zeichen, Worten, Zeilen   */
#define IMWORT  1      /* sprechende symbolische Konstante */
#define OUTWORT 0

void main()
{
    int zeich, nworte=0, nzeichen=0, nzeilen=0, status = OUTWORT;

    printf ("\nGeben Sie einen beliebigen Text ein (Ende STRG-Z) : \n");
    while ( (zeich = getchar() ) != EOF)
    {
        ++ nzeichen;
        if (zeich == '\n') ++nzeilen;
        if (zeich == ' ' || zeich == '\n' || zeich == '\t')
           status = OUTWORT;
        else
            if (status == OUTWORT) /* vom Wort zum Trennzeichen */
            {  status = IMWORT;
               ++nworte;
            }
    }
    printf ("\nZeichen : %d, Worte : %d, Zeilen : %d\n",
             nzeichen, nworte, nzeilen);
}
```

6.4. do while - die „until“ - Schleife in C

Die Bedingung wird am Schleifenende geprüft!

do Aktion while (Bedingung)



Wichtig

Zuerst wird die Aktion ausgeführt, dann wird die Bedingung geprüft. Ist diese TRUE, dann wird die Aktion erneut ausgeführt, usw. Wesentlich ist, daß die Aktion, im Gegensatz zur while Schleife, unabhängig von der Bedingung mindestens einmal durchlaufen wird - nur wenn dies sinnvoll und notwendig ist sollten Sie die do while Schleife einsetzen. Besteht die Aktion aus mehreren Statements, dann sind diese in {} einzuschliessen.

```
#include <stdio.h>        /* Beispiel für die do-while Schleife */
void main()
{
int i;

   i = 0;
   do
   {  printf("Der Wert von i ist jetzt %d\n",i);
      i = i + 1;
   } while (i < 5);                    /* Durchläufe: 0, 1, 2, 3, 4 */
}                                      /* i hat nacher den Wert 5   */
```

So könnte das Programm auch realisiert werden:

```
#include <stdio.h>
void main()            /* kurze C-like  Variante, gleiche Aktivität */
{   int i=0;
    do printf("Der Wert von i ist jetzt %d\n",i++);
    while (i < 5);
}
```

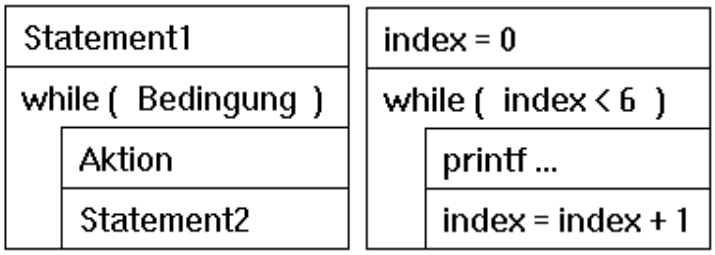
Dies ist ein Beispiel, bei dem der Schleifenkörper mindestens einmal durchgefürt werden muß, weil sonst die Bedingung nicht sinnvoll geprüft werden kann!

```
/* Beispiel für ein Programmstück, bei dem das Programm auf korrekte
   Eingabe eines numerischen Werts von 1 - 9 besteht */
ziffer = 0;
do
{   printf ("Geben Sie die Ziffer (1 - 9) ein : ");
    scanf  ("%d",&ziffer);              /* gesicherte Eingabeschleife */
    fflush (stdin);               /* zum Einlesen eines Werts   */
}
while (ziffer < 1 || ziffer > 9);
```

6.5. for - Schleife (iterative Schleife)

iterative oder zählende Schleifen sind while - Schleifen kombiniert mit anderen Statements

for (Statement1;
    Bedingung;
    Statement2)
    Aktion



```
/* ein simples for-Beispiel ähnlich dem obigen while Beispiel*/
#include <stdio.h>
void main()
{
int index;

   for(index = 0;index < 6;index = index + 1)
     printf("Der Wert der Variablen index ist momentan %d\n",index);
}
```

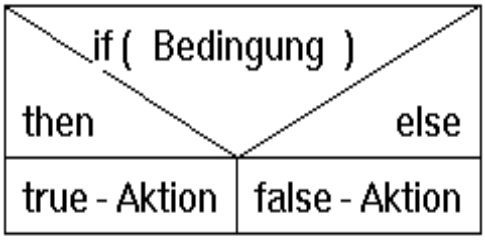
```
#include <stdio.h>
void main ()        /* Was macht wohl dieses nächste Beispiel ? */
{
  unsigned int   z,d,schalter_weiter;
  for (z=2;z<65535;++z)                      /* ++ erhöht z um 1     */
  {   schalter_weiter = 1;
      for (d=2;d < z && schalter_weiter;++d) /* && ist logisches und */
          if (z%d == 0) schalter_weiter=0;   /* wenn modulo z/d = 0  */
      if (schalter_weiter != 0)              /* !=  ist Ungleichheit   */
         printf("%u ist P...zahl \n",z); /* wenn ungleich dann drucke*/
  }
                          /* Die innere for Schleife enthält nur ein  */
}                         /*                if - Kommando als Aktion  */
```

Die for - Schleife bietet das oben angezeigte Grundkonstrukt für den Ablauf der einzelnen Teile. Jede der 3 Statementgruppen kann auch weggelassen werden.

for (i=0;;i++) ..... ist die zählende Endlosschleife mit i als Zähler

for (;;) ..... ist eine einfache Endlosschleife

6.6. if else (die klassische Bedingung)



Beim if gibt es kein Schlüsselwort then

if (bedingung) true_aktion
else false_aktion nur bei Bedarf

Wie auch bei while, etc. ist die Bedingung in runden Klammern zu schreiben. Ein then gibt es nicht, der true-Zweig folgt direkt auf die schließende Klammer. Im kürzesten Fall ist dieses Statement ein ; (leere Anweisung) oder irgend ein anderes Statement oder mehrere Statements in geschwungenen Klammern. Der else-Zweig ist nur bei Bedarf vorhanden, es gilt sinngemäß dasselbe wie beim true-Zweig.

```
/* Das ist ein Beispiel für if und if-else Statements */
#include <stdio.h>
void main()
{
   int data;

   for(data = 0;data < 10;data = data + 1)
   {
      if (data == 2)          /* == ist der Vergleichsoperator gleich */
         printf("data enthält jetzt %d\n",data);

      if (data < 5)
```

```
         printf("data ist jetzt %d, kleiner als 5\n",data);
      else
         printf("data ist jetzt %d, was größer als 4 ist\n",data);
   }  /* Ende des for Loop */

}              /* überlegen Sie wie die Schleife ausgeht, wenn Sie */
               /* statt   data == 2   nur   data = 2  schreiben ?? */
```

Bei geschachtelten if else Konstrukten muß man wie üblich darauf achten, daß die else Zweige auch zu den richtigen ifs gehören. Es gilt:

+ das else gehört zum jeweils letzten if.

```
          /* falsch ist folglich : */
if (Bedingung-1)
   if (Bedingung-2) aktion im if-2 .......... ;
else aktion im else-1 ............... ;
          /* richtig wird es so gemacht : */
if (Bedingung-1)
   if (Bedingung-2) aktion im if-2 .......... ;
   else ;
else aktion im else-1 ............... ;
          /* man könnte aber auch folgendes versuchen : */
if (Bedingung-1)
{  if (Bedingung-2) aktion im if-2 .......... ;
}
else aktion im else-1 ............... ;
```

6.7. break und continue (aussteigen aus Schleifen)

break ist manchmal sinnvoll, continue sollte man meiden.

break bewirkt den sofortigen Ausstieg aus while, do while und for Schleifen und auch aus dem switch - Statement. continue bewirkt den nächsten Durchlauf einer while, do while oder for Schleife; es werden dabei die restlichen zu "Aktion" gehörenden Statements übersprungen. Bei mehreren geschachtelten Schleifen wirken break und continue nur für die Schleife, in der sie verwendet werden. Ein break in der innersten von drei geschachtelten Schleifen verläßt nur die innerste, aber nicht etwa alle drei Schleifen.

```
#include <stdio.h>
void main()
{
int xx;

   for(xx = 5;xx < 15;xx = xx + 1)
   {  if (xx == 8)
         break;
      printf("In der break Schleife ist xx jetzt %d\n",xx);
   }
   for(xx = 5;xx < 15;xx = xx + 1)
   {  if (xx == 8)
         continue;
      printf("In der continue Schleife ist xx jetzt %d\n",xx);
   }
}
```

Bei der ersten Schleife gibt printf daher die Werte 5,6,7 aus. Beim Wert 8 erfolgt ein Ausstieg aus der Schleife. In der zweiten Schleife erscheinen aber die Werte 5,6,7,9,10,11,12,13,14. Die Ausgabe von 8 wird durch continue übersprungen.

6.8. switch (die C Variante des Case Befehls)

switch prüft einen Ausdruck auf Gleichheit mit unterschiedlichen Werten

Mit einer if else if else if else .... Kette kann eine Aneinanderreihung von Bedingungen erfolgen, wobei die Sequenz solange durchlaufen wird, bis ein if true wird. Dieser spezielle Fall sollte laut strukturierter Programmierung mit einer speziellen Anweisung erledigt werden können. Leider ist die switch Anweisung nur in der Lage auf unterschiedliche ganzzahlige Werte einer Variablen bzw. eines Ausdrucks zu reagieren, aber nicht auf eine beliebige Bedingungskette. Die Variable muß auch int oder char sein.

```
switch (Bedingungsausdruck)
{
   case  Wert1 : Aktion1;
   case  Wert2 : Aktion2;
   default  : Aktion-n;
}
```

Der default Zweig kann weggelassen werden. Auch wenn die Aktionen aus mehr als einem Statement bestehen, bedarf es keiner geschwungenen Klammer. Dafür steht der gesamte Statementteil nach "(Bedingungsausdruck)" in {}.

Achtung: Es ist üblich, jede Aktion nach case oder default mit einem break zu beenden, weil ansonsten ab dem Zutreffen eines case Zweiges alle Aktionen bis zum Ende des switch - Statements ausgeführt werden (also jene der nachfolgenden case Teile und des default Teils). Dies ist eine etwas gewöhnungsbedürftige C Spezialität; vergessen Sie das break nicht!

beim switch sind breaks notwendig !

```
#include <stdio.h>
void main()
{
int lauf;

   for (lauf = 3;lauf < 13;lauf = lauf + 1)
   {
      printf ("lauf=%2d -- ",lauf);
      switch (lauf) {
         case 3  : printf("Der Wert ist jetzt drei\n");
                   break;
         case 4  : printf("lauf enthält jetzt 4\n");
                   break;
         case 5  :
         case 6  :
         case 7  :
         case 8  : printf("Wert ist im Bereich von 5 bis 8\n");
                   break;   /* Alle Werte von 5 - 8 bringen diese  */
                            /* Meldung, weil kein break bei 5,6,7  */
         case 11 : printf("Jetzt schlägt es elf\n");
                  break;
         default : printf("ein Wert ohne besonderen Text\n");
                   break;
      } /* Ende zu switch */
   } /* Ende for Schleife */
}
```

Der Weg eines Programmieranfängers bis zum Meister, der in der Lage wäre heutige professionelle Programme zu schreiben (alle heutigen Programme werden aber in großen Teams und nicht mehr von genialen Einzelgängern geschrieben) ist lang. Daher ist die Bezeichnung simples Menüsystem für das folgende Programm durchaus gerechtfertigt, obwohl es sicher keinem heute verwendeten Menü nahekommt.

```
#include <stdio.h>
#include <conio.h>

void main ()      /* simples Menüprogramm, zeigt Ausgabe mit conio.h */
{                 /* Eingabe mit getch (siehe Sondertasten) und eine */
   int taste=0;   /* Anwendung des switsch Befehls.                  */

   while (taste != 'E')
   {
      clrscr();          /* der Menü - Ausgabetext */
      gotoxy(10,3);
      printf ("Minibeispiel eines Menüsystems");
      gotoxy(14,6);
      printf ("A - Menüpunkt A auswählen ");
      gotoxy(14,8);
      printf ("B - Menüpunkt B auswählen ");
      gotoxy(14,10);
      printf ("C - Menüpunkt C auswählen ");
      gotoxy(14,18);
      printf ("E - Programm beenden");
      gotoxy(45,20);
      printf ("bitte auswählen ");
      gotoxy(65,20);

      taste = getch();  /* Tastendruck lesen, wenn 0, dann ist es eine  */
      if (taste == 0) taste = getch () + 256; /* Funktions-/Cursortaste */
          /* 0-255  sind Codes normal gedrückter Tasten,
             > 256 sind nun die Codes der Spezialtasten */
       /* die folgende Anzeige zum besseren Verständnis */
             printf("%d",taste);
             if (taste < 256) printf (" %c",taste);


      gotoxy (5,24);
      switch (taste)    /* Auswerten der gedrückten Taste */
      {
       case 'a' :
       case 'A' :  printf ("Menüpunkt A noch nicht implementiert");
                   break;
       case 'b' :
       case 'B' :  printf ("Menüpunkt B noch nicht implementiert");
                   break;
       case 'c' :
       case 'C' :  printf ("Menüpunkt C noch nicht implementiert");
                   break;
       case 'e' :
       case 'E' :  taste = 'E';
                   break;
       default  :  gotoxy (5,24);
                   printf ("falsche Taste - wählen Sie einen Menüpunkt");
      }
      delay (2000); /* 2 sec warten - damit man Fehlermeldung lesen kann */
   }
      printf ("Bye\n\n");
}
```

6.9. goto (ja auch dieses Statement existiert)

Wo ein goto existiert, muß es auch Labeln geben, die mit dem goto angesprungen werden. Beides ist in C möglich. Weil alle notwendigen Statements für die Realisierung strukturierter Programme vorhanden sind, gibt es wenig Grund einen goto zu verwenden. Das folgende Beispielprogramm ist zwar ein abschreckendes Beispiel, zeigt aber gleichzeitig die wahrscheinlich einzige sinnvolle Nutzung des goto Statements.

kein Vorbild !

```
#include <stdio.h>
void main()
{
int hund,ochse,schwein;

   goto start_ist_hier;
   irgendwo:
   printf("Eine andere Zeile aus diesem Mist.\n");
   goto genug_ists;
/* in der innersten Schleife steht der einzige sinnvolle goto */
   start_ist_hier:
   for(hund = 1;hund < 6;hund++) {
      for(ochse = 1;ochse < 6;ochse++) {
        for(schwein = 1;schwein < 4;schwein++) {
            printf("Hund = %d  Ochse = %d  Schwein = %d\n",
                    hund,ochse,schwein);
            if ((hund + ochse + schwein) > 8 ) goto es_reicht;
         };
      };
   };
   es_reicht: printf("Für\'s erste sind das genug Viecher.\n");

   printf("\nDas ist die erste Zeile vom Spaghetti Code\n");
   goto dort;

   im_wald:
   printf("Die dritte Zeile vom Spaghetti Code.\n");
   goto irgendwo;

   dort:
   printf("Es folgt das zweite Spaghetti.\n");
   goto im_wald;

   genug_ists:
   printf("Ende des ");
   printf ("Wartungsprogrammiererarbeitsplatzsicherungsprogramms .\n");

}
```

6.10. return (Beenden Unterprogramm, zurückgeben Funktionswert)

Unterprogramme und Funktionen werden genauso geschrieben wie die Funktion main, die bereits in vielen Beispielen vorkam. Das Ende einer Funktion ist mit der letzten schließenden } gegeben. Will man aber schon vorher ein Unterprogramm verlassen oder muß man (bei Funktionen) einen Wert zurückgeben, so geht das mit return bzw. mit return Ausdruck.

```
int sign (int x)/*  das ist eine Funktion, die das Vorzeichen des */
                /*  Parameters bestimmt; x ist der Parameter      */
{
 if (x > 0)
      return (1);
 else if (x < 0)
           return (-1);
     else return (0);
}
```

Als Hauptprogramm sind die folgende Zeilen sinnvoll:

Prototypen verringern die Fehlergefahr bei Funktionsaufrufen wesentlich

Wichtig

```
#include <stdlib.h>
#include <stdio.h>

int sign(int); /* vor der 1. Funktion steht diese Zeile, der sogenannte
                  "Prototyp"; dadurch kennt der Compiler Anzahl und Daten-
                  typ der Parameter und des Funktionswerts und kann die
                  später im Code vorkommenden Aufrufe überprüfen */
void main ()
{   int i,j,vorzeichen;
    while ( (j = scanf("%d",&i)) != EOF )
    {   /* in die Variable i wurde nun eine Zahl eingelsen */
        vorzeichen =    sign(i);    /* Aufruf der Funktion */
        printf ("Das Vorzeichen von %d ist %d \n",i,vorzeichen);
        fflush (stdin);
    }
}
```

Damit wird eine selbstgeschriebene Funktion aufgerufen, mit printf wurde bereits öfter eine Funktion aus den Standardbibliotheken verwendet.

Weil hier erstmals Parameter verwendet werden, muß auch die wahrscheinlich wichtigste Ansi C Neuerung, die unterschiedliche Parameter Notation, erläutert werden. Ansi C versteht auch die alte Schreibweise.

```
Altes C                                          Ansi C

int sign (x)                                int sign (int x)
int x;  /* Parameter extra */
{ .........                                 { ..............
}                                           }

/* ---- im Hauptprogramm steht als Prototyp      */

int sign();                                 int sign (int);

/* ---- im alten C wurde dem Compiler des Hauptprogramms nichts über die
        Parameter mitgeteilt !!  */
```

Weil nun die Befehle fertig aufgezählt sind, wird wieder erinnert, daß diese auch übersichtlich angewendet werden sollen. Es folgen hiezu gute und schlechte Beispiele.

Besonders bei kurzen Beispielprogrammen neigt man dazu, das Programm eher unschön zu schreiben. So ist das folgende Programm zwar gut eingerückt, aber das ist auch schon alles.

```
#include <stdio.h>
void main()
{
int x1,x2,x3;

   printf("Temperaturtabelle Celsius und Farenheit\n\n");

   for(x1 = -2;x1 <= 12;x1 = x1 + 1)
   {  x3 = 10 * x1;
      x2 = 32 + (x3 * 9)/5;
      printf("  C =%4d   F =%4d  ",x3,x2);
      if (x3 == 0)
         printf(" Gefrierpunkt des Wassers");
      if (x3 == 100)
         printf(" Siedepunkt des Wassers");
      printf("\n");

   }
}
```

Obwohl das Programm alles tut, was die Problemstellung verlangt, kann man damit nicht zufrieden sein. Nur weil zufällig drei längere Strings gedruckt werden, kann man leicht erraten was das Programm macht. Es fehlen sprechende Namen aber auch jegliche Dokumentation und Kommentierung. Dabei wäre die Lösung auch wesentlich besser denkbar; beispielsweise so:

Eine einfache und effektive Dokumentation am Beginn des Programmfiles

```
/*Name**********Kurzbeschreibung*********************************/
/* temptab.c   * dieses Programm listet eine Temperaturtabelle  */
/*Cr-Datum****** Celsius - Fahrenheit (in 10er Schritten von    */
/* 1.9.92      * -20 bis 120 °C) auf stdout                     */
/*Funktionsaufrufe************************************************/
/* keine eigenen Funktionen                                    */
/* C-Funktionen : printf                                       */
/*Dateien*********************************************************/
/* keine Dateien in Verwendung                                 */
/*Parameter*******************************************************/
/* keine Parameter, Rückgabeparameter oder Kommandozeilenangaben*/
/*Exit-Status/Rückgabewert****************************************/
/* wird nicht gesetzt (kein  Exit-Code ans Betriebssystem )     */
/* in Funktionen würde hier der Rückgabewert beschrieben        */
/*globale Variable************************************************/
/* keine Verwendung oder Änderung globaler Variablen           */
/*Beschreibung****************************************************/
/* hier wird eine längere Beschreibung (Grobablauf in Pseudo-   */
/* code, Übersichtstabellen, etc.) erwartet                    */
/*Aenderungen*****************************************************/
/* hier sollten spätere Änderungen mit Datum und Text stehen    */
/*****************************************************************/

#include <stdlib.h>
#include <stdio.h>
void main()
{                /* Kommentar zu wichtigen Variablen und Codestellen */
int i;
int fahrenheit;   /* enthält Temperatur in Fahrenheit Graden */
int celsius;      /* enthält Temperatur in Celsius   Graden  */

   printf("Temperaturtabelle Celsius und Fahrenheit\n\n");

   for(i = -2;i <= 12;i++)               /* Schleife pro Zeile der */
   {                                      /* Ausgabetabelle         */
      celsius = 10 * i;                   /* mal 10, für die Ausgabe */
                                          /* in Zehnersprüngen       */
      fahrenheit = 32 + (celsius * 9)/5;  /* Umrechnungsformel       */
      printf("  C =%4d   F =%4d  ",celsius,fahrenheit);
      if (celsius == 0)
         printf(" Gefrierpunkt des Wassers");
      if (celsius == 100)
         printf(" Siedepunkt des Wassers");
      printf("\n");
   } /* Ende for Schleife */
} /* Ende Funktion main */
```

Es folgt ein weiteres Beispiel:

```
/*Name**********Kurzbeschreibung*********************************/
/* name        * Dieses Programm zeigt alle Zahlen von 0 - 99,  */
/*Cr-Datum****** die ein besonderes Nahverhältnis zu einer      */
/* 1.9.92      * bestimmten Ziffer haben.                       */
/*Funktionsaufrufe************************************************/
/* keine eigenen Funktionen                                    */
/* C-Funktionen : printf                                       */
/*Dateien*********************************************************/
/* keine Dateien in Verwendung                                 */
/*Parameter*******************************************************/
/* keine Parameter, Rückgabeparameter oder Kommandozeilenangaben*/
/*Exit-Status/Rückgabewert****************************************/
/* wird nicht gesetzt (kein  Exit-Code ans Betriebssystem )     */
/* in Funktionen würde hier der Rückgabewert beschrieben        */
/*globale Variable************************************************/
```

```
/* keine Verwendung oder Änderung globaler Variablen            */
/*Beschreibung***************************************************/
/* Nach einer Programmerklärung muß der Benutzer die Ziffer ein-*/
/* geben. Dann werden als 10 x 10 Tabelle die Zahlen von 0 - 99 */
/* ausgegeben, wobei die zur Ziffer verwandten Zahlen durch     */
/* Striche ersetzt werden.                                      */
/* Zahlenverwandtschaft ergibt sich durch :                     */
/*  - Einerstelle der Zahl gleich der Ziffer                    */
/*  - Zehnerstelle der Zahl ist gleich der Ziffer               */
/*  - die Quersumme der Zahl ist gleich der Ziffer              */
/*  - die Zahl ist durch die Ziffer (ohne Rest) teilbar         */
/*Aenderungen****************************************************/
/* hier sollten spätere Änderungen mit Datum und Text stehen    */
/****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#define CODE_STRICH '\xdd'

void main ()
{
    int  i,ziffer,einer,zehner,quersumme,rest;
    char strich=CODE_STRICH; /* dies ist ein semigraphischer Strich */

    printf ("Selektiere Zahlen (0-99), die eine bestimmte Ziffer \n");
    printf ("enthalten, deren Quersumme die Ziffer ist bzw. die  \n");
    printf ("durch die Ziffer (ohne Rest) teilbar sind.          \n\n");

    do
    {     printf ("Geben Sie die Ziffer (2 - 9) ein : ");
          scanf  ("%d",&ziffer);         /* gesicherte Eingabeschleife */
          fflush (stdin);                /* zum Einlesen eines Werts   */
    }
    while (ziffer < 2 || ziffer > 9);

    for (i=0; i<100; i++)               /* Schleife von 0 - 99      */
    {     einer     = i % 10;           /* Einerstelle ist der Rest */
          zehner    = i / 10;           /* ganzzahlige Division         */
          quersumme = einer + zehner;
          rest      = i % ziffer;

          if (einer == 0) printf ("\n"); /* 10 Zahlen pro Zeile
                                            ergeben eine Zahlentabelle */
          if (einer     == ziffer ||     /* prüfe auf die geplanten */
              zehner    == ziffer ||     /* Kriterien                */
              quersumme == ziffer ||
              rest      == 0)
               printf ("  %c%c",strich,strich); /* ja,   also Strich  */
          else printf ("%4d",i);                /* nein, also Zahl    */
    }
    printf ("\n\n");                        /* Zeilenvorschub am Ende   */
}
```

Wird ein Testlauf durchgeführt und dabei die Ziffer 7 eingegeben, so erscheint die folgende Ausgabe:

```
 ||   1   2   3   4   5   6  ||   8   9
 10  11  12  13  ||  15  ||  ||  18  19
 20  ||  22  23  24  ||  26  ||  ||  29
 30  31  32  33  ||  ||  36  ||  38  39
 40  41  ||  ||  44  45  46  ||  48  ||
 50  51  ||  53  54  55  ||  ||  58  59
 60  ||  62  ||  64  65  66  ||  68  69
 ||  ||  ||  ||  ||  ||  ||  ||  ||  ||
 80  81  82  83  ||  85  86  ||  88  89
 90  ||  92  93  94  95  96  ||  ||  99
```

Nun folgt noch ein letztes Beispiel, bevor es mit C weitergeht:

Die Ansi - genormten ESC - Sequenzen sind ein taugliches Mittel zur Bildschirmansteuerung

```
/* Das nachfolgende Beispiel zeigt den möglichen Einsatz einer  */
/* Funktion. Das Hauptprogramm hat nur den Zweck die Funktion   */
/* zu testen. Wie man sieht zeichnet die Funktion einen Rahmen  */
/* auf den Bildschirm. Die Rahmenposition und die Größe wird    */
/* als Parameter an die Funktion übergeben.                     */
/*                                                              */
/* Dieses Programm ist nicht direkt portabel, durch Anpassung   */
/* der #define Zeilen kann es aber an einen anderen Bildschirm  */
/* angepaßt werden. Das Programm läuft aber immer nur auf einem */
/* bestimmten Bildschirm. Das ist am PC brauchbar, für Unix     */
/* Maschinen aber nicht denkbar.                                */

#include <stdlib.h>
#include <stdio.h>
#include <conio.h>      /*  wegen der getch - Funktion */

#define C_CLS             \033[2J          /* clear screen        */
#define C_GOTOXY          \033[%d;%dH      /* positioniere Cursor */
#define CLS               printf("\033[2J")
#define GOTOXY(z,s)       printf("\033[%d;%dH",z,s)

#define C_STRICH          "-|abcd"                /* Rahmen: a---c  */
                                                  /*         |   |  */
                                                  /*         b---d  */
int rahmen (int,int,int,int); /* Prototyp der Funktion */

void main ()
{

    int    i,lz,ls,nz,ns;

    CLS;
    printf ("Geben Sie bitte die notwendigen Koordinaten für die \n");
    printf ("Erstellung eines Rahmens ein.                       \n");
    printf ("Zeile  links oben : ");
    scanf  ("%d",&lz); fflush(stdin);
    printf ("Spalte links oben : ");
    scanf  ("%d",&ls); fflush(stdin);
    printf ("Anzahl Zeilen    : ");
    scanf  ("%d",&nz); fflush(stdin);
    printf ("Anzahl Spalten   : ");
    scanf  ("%d",&ns); fflush(stdin);

    CLS;
    i = rahmen(lz,ls,nz,ns);
    getch(); /* diese Funktion liest nur eine Taste (auch ohne return) */
}

int rahmen (int links_zeile,int links_spalte,
         int anzahl_zeilen,int anzahl_spalten)
{
   int i,fehler = 0;
   static char chrahmen[7]= C_STRICH;

   if (links_zeile   > 22 || links_zeile   < 0) {
      links_zeile   = 1;
      fehler = 1;
   }
   if (links_spalte  > 78 || links_spalte  < 0) {
      links_spalte  = 1;
      fehler = 1;
   }
   if (anzahl_zeilen + links_zeile > 22 || anzahl_zeilen  < 0) {
      anzahl_zeilen = 0;
      fehler = 1;
   }
   if (anzahl_spalten + links_spalte > 78 || anzahl_spalten < 0) {
      anzahl_spalten = 0;
      fehler = 1;
   }

   CLS;
```

```
   GOTOXY(links_zeile,links_spalte);
   printf ("%c",chrahmen [2]);
   for (i=1;i <= anzahl_spalten;i++) printf ("%c",chrahmen[0]);
   printf ("%c",chrahmen [4]);
   for (i=1;i <= anzahl_zeilen;i++) {
       GOTOXY(links_zeile+i,links_spalte);
       printf ("%c",chrahmen[1]);
       GOTOXY(links_zeile+i,links_spalte+anzahl_spalten+1);
       printf ("%c",chrahmen[1]);
   }
   GOTOXY(links_zeile+anzahl_zeilen+1,links_spalte);
   printf ("%c",chrahmen [3]);
   for (i=1;i <= anzahl_spalten;i++) printf ("%c",chrahmen[0]);
   printf ("%c",chrahmen [5]);

   return fehler;
}
```

Es wird Ihnen wohl unangenehm auffallen, daß die Funktion keine Dokumentation und auch keinen Kommentar enthält, lernen Sie daraus!

Übungen :

- Schreiben Sie ein Programm, welches Textzeilen einliest und mit einer Zeilennummerierung versehen wieder ausgibt.

- In vielen Texten sind Tabulatoren enthalten. Schreiben Sie ein Programm, welches Text einliest und die darin enthaltenen Tabulatoren in Leerzeichen (1-8 Leerzeichen anstelle eines \t) umwandelt. Als Tabulatorenschrittweite wird 8 angenommen.

- Lesen Sie eines Ihrer C - Programme ein. Berechnen Sie den „Kommentarkoeffizienten“ - das Verhältnis Zeichen_im_Programm / Zeichen_innerhalb_von_Kommentar und geben Sie dies möglichst informativ aus.

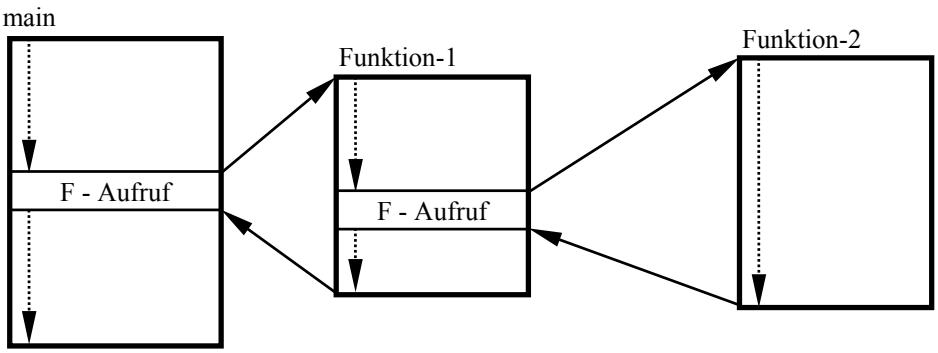
7. Der Aufbau eines C - Programms

7.1. Funktionen und Unterprogramme allgemein

Bislang bestand unser Programm immer aus der Funktion main, darin wurden alle Variablen definiert und die eigentlichen Programmbefehle geschrieben. Die Programmbefehle werden sequentiell hintereinander bzw. gemäß dem Ablauf von Befehlen wie if, while, switch abgearbeitet. Abgesehen davon, daß Codeteile ausgelassen (if) oder andere Codeteile wiederholt ausgeführt werden (while, for) folgt die Programmabarbeitung im Groben der Reihenfolge von oben nach unten durch die Funktion main. Unsere bisherigen Problemlösungen waren eher klein, für ernsthafte Probleme müßte man aber den Code und damit die Funktion main wesentlich vergrößern. An einer Funktion kann aber praktisch nur eine Person arbeiten, folglich wären große Softwareprojekte niemals denkbar.

Die gesamte Aufgabenstellung wird daher möglichst sinnvoll in Teilbereiche aufgeteilt, Module werden gebildet. Bei größeren Softwareprojekten ist genau diese Modularisierung die anspruchsvollste und schwierigste Programmieraufgabe, seien Sie froh, daß Sie vorläufig nur mit kleinen Programmieraufgaben befaßt sind. Jeder Teilbereich ist nun eine eigenständige überschaubare Einheit, die auch arbeitsmäßig bewältigbar ist.

Aus der Sicht einer Programmiersprache steckt der gesamte Programmcode nicht in der Funktion main, sondern wird auf mehrere (bei größeren Aufgaben auch viele) Funktionen aufgeteilt. Der Programmablauf beginnt nach wie vor in main, beim Aufruf einer Funktion bzw. eines Unterprogramms wechselt die Programmausführung aber dorthin, arbeitet die Funktion ab und kehrt wieder zur rufenden Funktion (main) zurück, wo die Ausführung weitergeht. Zu den normalen Befehlen der Programmiersprache (if, while, ...) kommt zusätzlich die Möglichkeit andere Funktionen aufzurufen und deren Funktionalität zu verwenden. Die folgende Abbildung skizziert die dann stattfindende Reihenfolge der Programmabarbeitung.



Obwohl bei der Ausführung der ganze Programmcode durchlaufen wird kann man die Codeerstellung problemlos auf verschiedene Personen aufteilen, die voneinander fast nichts wissen müssen. Der Schreiber der rufenden Funktion muß die Spezifikation (Eingangsgrößen, Ausgangsgrößen, Zweck) der aufgerufenen Funktion kennen. Der Autor der aufgerufenen Funktion muß dafür sorgen, daß seine Funktion getreu den Spezifikationen funktioniert und auch durch falsche Eingangsgrößen nicht abstürzt oder sonstigen groben Unfug macht. Es ist daher auch üblich, daß vorhandene (gut dokumentierte) Funktionen lange nach ihrer Erstellung noch immer von den Autoren neuer Programme verwendet werden.

Eine Funktion sollte

- einen gemäß der gesamten Aufgabenstellung logisch zusammengehörigen Programmabschnitt umfassen.
- mehrfach verwendbar sein, also von verschiedenen Stellen im restlichen Programm aufgerufen werden. Wiederkehrende Programmteile werden nur einmal codiert.
- klein und kompakt bleiben (Hausregel: nicht wesentlich über eine Seite A4) - ist die Funktion zu groß, dann muß sie in weitere Funktionen aufgeteilt werden.
- eine minimale Schnittstelle (Parameter, Funktionsrückgabewert) zum restlichen Programm aufweisen.
- in sinnvoller Kombination mit den anderen Funktionen den Gesamtcode des Programms minimieren.

Auch die Erfinder der Sprache C waren von der Sinnhaftigkeit von Funktionen überzeugt und haben massiv davon Gebrauch gemacht. In C gibt es nur sehr wenig wirkliche Befehle, die vom Compiler auch als Befehl interpretiert und in Maschinencode umgesetzt werden. Alle echten Befehle (if, while, for) sind in der Liste der gesperrten Wörter. Ist ein „Befehl" dort nicht dabei (getchar, printf, strcpy, ...) dann ist er eigentlich eine Funktion und wurde mit dem Compiler mitgeliefert. Wenn Ihr Programm die Zeilen

```
char name[80] = {"Rosalinde"};
...........
printf ("Der Name ist %d Stellen lang\n",strlen(name));
```

enthält, dann sind das zwei Funktionsaufrufe, printf und strlen. Zuerst wird strlen aufgerufen. Das char - Array name stellt die Eingangsgröße von strlen dar. Nach der Ermittlung der Länge wird strlen diese als Zahl zurückliefern, die wiederum in printf als Eingangsgröße Verwendung findet. Während printf eine komplizierte Funktion ist, könnte strlen etwa so aussehen:

```
int strlen (const char stri[])  /* Beispiel, ermittelt Länge eines Strings
*/
{
   int i;
   for (i = 0; stri[i] != '\0'; i++) ;
   return (i);
}
```

Die erste Zeile legt durch **int** fest, daß strlen einen Funktionsrückgabewert vom Typ int zurückliefern wird. **(char stri[])** beschreibt einen Parameter (die Eingangsgröße), ein char-Array. Die eckigen Klammern sind deshalb leer, weil das Array immer die Größe jenes Arrays annimmt, das beim Funktionsaufruf mitgegeben wird. Als letztes Statement der Funktion findet sich **return(i)**, wodurch der errechnete Wert zurückgegeben wird.

Bei **void main()** ist das Fehlen von Parametern leicht festzustellen. Durch **void** wird zusätzlich festgelegt, daß es auch keinen Funktionsrückgabewert gibt. Funktionen ohne Rückgabewert nennt man Unterprogramme (auch Proceduren bzw. Subroutinen).

In C sind Unterprogramme also als Spezialfall einer Funktion möglich, in den meisten anderen Sprachen werden Unterprogramme durch den Befehl CALL aufgerufen. Bei Funktionen wird hingegen immer der Funktionsrückgabewert genutzt, Funktionen werden daher so verwendet als wären sie Variable. Einer Variablen kann man allerdings auch Werte zuweisen, bei Funktionen ist das nicht möglich.

Den einfachsten Fall eines Unterprogramms zeigt das folgende Beispiel

```
void updruck()     /* Unterprogramm welches eine Textzeile ausgibt */
{
    printf (" und UP-Zeilenabschluß ausgeben \n");
}
```

```
........
void main ()
{
    printf ("Zeile 1 ausgeben ");    /* updruck wird mehrfach aufgerufen */
    updruck();                       /* die Aktivität ist jedesmal genau */
    printf ("Zeile 2 ausgeben ");    /* gleich                           */
    updruck();                       /* es gibt keinen Datenaustausch    */
    printf ("Zeile 3 ausgeben ");    /* zwischen main und updruck        */
    updruck();
    printf ("letzte Zeile ausgeben \n");
}
```

In diesem Fall wird eine mehrfach vorkommende Codezeile aus der Funktion main ausgelagert und durch den Unterprogrammaufruf ersetzt. Es werden aber keinerlei Daten von/nach updruck übergeben. Weil diese Form wenig flexibel ist ist es üblich zumindest Parameter zu verwenden.

```
void quadrate(int zahl)  /* Unterprogramm das zahl * zahl ausgibt */
{
    printf ("    n = %5d, n² = %d\n",zahl,zahl * zahl);
}
........
void main ()
{
    printf ("Quadrate von Zahlen ausgeben : \n");
    quadrate (3);                   /* updruck wird mehrfach aufgerufen */
    quadrate (9);                   /* bei jedem Aufruf ist die Zahl    */
    quadrate (17);                  /* anders - das Upro kann wesentlich*/
    quadrate (33);                  /* flexibler eingesetzt werden      */
}
```

Das Unterprogramm quadrate hat einen Parameter, der den Eingangswert liefert, es werden keine Daten vom Uinterprogramm zurück an main geliefert. Unterprogramme haben häufig mehrere Parameter, damit können Werte vom aufrufenden Programmteil (Hauptprogramm) zum aufgerufenen Programmteil (Unterprogramm) übertragen werden. Die Möglichkeit der Datenrückübertragung existiert auch, aber nicht in allen Fällen.

Bei Funktionen gibt es neben den Parametern den Funktionsrückgabewert, der das eigentliche Wesen einer Funktion darstellt, dadurch können Funktionsaufrufe auch mitten in Ausdrücken (Berechnungen) verwendet werden. Eine Quadratwurzel wird so verwendet :

```
c = sqrt(a*a + b*b);
```

7.2. Schnittstelle zwischen Funktionen (Funktionsaufruf)

Verbindung zwischen Modulen = **Schnittstelle**

Zwischen Haupt- und Unterprogramm bzw. zwischen rufender und augerufener Funktion gibt es die folgenden Möglichkeiten Daten auszutauschen:

7.2.1. Parameter

Parameter übergeben Werte vom rufenden Programm zur Funktion (und manchmal auch zurück)

Parameter können beim Aufruf an Funktionen bzw. Unterprogramme übergeben werden. Im Gegensatz zu anderen Programmiersprachen wie Cobol oder Fortran wird aber nicht Call by Adress sondern Call by Value verwendet. Das heißt, die Parameter werden im Hauptprogramm in temporäre Felder (in C in den sogenannten Stack) gestellt und dann diese Felder (deren Adressen) ans Upro (Funktion) übergeben. Dort können die (formalen) Parameter beliebig verändert werden, es werden aber bei der Rückkehr ins Hauptprogramm die Parameter **NICHT** mehr zurück übernommen. Es kann also kein (aktueller) Parameter des Hauptprogramms durch ein Unterprogramm verändert werden. Ausnahme: Arrays (nur Name des Array, ohne Indexangabe) werden immer mit Call by Adress übergeben. Auch kann man Adressübergabe durch die aus C++ stammende Referenzschreibweise erreichen.

Parameter werden vom Hauptprogramm ins Unterprogramm übergeben aber NICHT mehr zurück übernommen. --> Call by Value

Wichtig

Das folgende Programm funktioniert deshalb nicht so wie man es erwarten könnte.

```
#include <stdio.h>
void tausch (int,int);

main ()
{
   int x=1,y=2;

   printf (" vor dem Tausch x=%d  y=%d\n",x,y);
   tausch (x,y);
   printf ("nach dem Tausch x=%d  y=%d\n",x,y);

}
void tausch (int a,int b)  /* soll a und b austauschen */
{
   int help;

   help = a;    /* hier wird zwar fleißig getauscht, das Haupt- */
   a = b;       /* programm bleibt aber völlig unbeeinflußt    */
   b = help;
}
```

In C hätte man in solchen Fällen noch mit Adressen (Pointer) hantieren müssen um händisch die Adressübergabe der Parameter zu erledigen. Bei der scanf Eingabefunktion sieht man das noch, denn man muß ein & vor alle Parameter schreiben, sonst könnten sie nicht verändert werden., somit könnte nichts eingelesen werden.

Wir wollen aber ein Element aus C++ verwenden, die sogenannte Referenz. Wir müssen nur & bei der Funktionsdefinition von tausch einfügen, das ganze als C++ Programm kompilieren (*.cpp) und schon haben wir **Call by Adress (**heißt hier **Referenzübergabe)**!

Wichtig

```
#include <stdio.h>
void tausch ( int &, int &);

main ()
{
   int x=1,y=2;

   printf (" vor dem Tausch x=%d  y=%d\n",x,y);
   tausch (x,y);
   printf ("nach dem Tausch x=%d  y=%d\n",x,y);

}
void tausch (int &a,int &b)  /* tauscht a und b aus */
{
   int help;

   help = a;    /* hier wird fleißig getauscht, das Haupt- */
   a = b;       /* programm merkt es diesmal auch    */
   b = help;
}
```

7.2.2. Funktionsrückgabewert

der **Funktionsrückgabewert**, das Wesen einer Funktion !

Funktionen geben beim Aufruf einen Funktionswert retour. Der typische Aufruf von Funktionen ist daher immer in einen Ausdruck eingebettet, wo mit dem Funktionsergebnis weitergearbeitet wird. Es ist aber auch möglich Funktionen aufzurufen, ohne den Funktionswert zu nutzen. Auch printf ist kein Unterprogramm, sondern eine Funktion; der Funktionswert ist vom Typ int und gibt die Anzahl der ausgedruckten Zeichen oder im Fehlerfall einen negativen Fehlercode zurück. Meist wird printf aber wie ein Unterprogramm genützt.

7.2.3. globale Variable

Normalerweise sind Variable lokal, das heißt nur innerhalb der Funktion verwendbar, in der sie definiert sind, es gibt aber auch globale (in C extern definierte) Variable, die in mehreren Funktionen verwendbar sind.

globale Variable sind in mehreren Funktionen bekannt

Extern definierte Variable, die in den Funktionen mittels "extern" deklariert werden. (In jenen Funktionen, die gemeinsam mit der extern-Definition compiliert werden, kann die Deklaration entfallen.) Diese extern (außerhalb einer Funktion) definierten Variablen stehen dann in jeder Funktion zur Verfügung, in der ein "extern" Verweis existiert und können daher auch jederzeit verändert oder abgefragt werden.

Globale (externe) Variable sollten sparsam angewendet werden. Je mehr globale Variable Sie einsetzen, desto mehr sind die einzelnen Funktionen voneinander abhängig. Allgemein sollten diese Variablen Werte enthalten, die global von Bedeutung sind, die sich aber während des Programmablaufs wenig oder gar nicht ändern.

Im nächsten und übernächsten Abschnitt werden lokale und globale Variable näher behandelt.

7.3. Funktionen und Sourcecodedateien in C

Generell werden in C einzelne Funktionen (Unterprogramme) geschrieben. Diese Funktionen können nicht ineinander geschachtelt werden, sondern nur in einem File (= eine Datei) hintereinander angeführt werden oder auch jeweils alleine in einem File stehen. Die Files (mit einer oder mehreren Funktionen) werden dann vom Compiler kompiliert; es entstehen die Objektmodule. Alle Objektmodule, die ein ausführbares Programm bilden sollen werden dann vom Linker zum ausführbaren Programm zusammengebunden. Dabei werden auch noch Bibliotheken für die Systemfunktionen verwendet.

Ein Datei eines C Programms (kann eine oder alle Funktionen eines Programms enthalten) sieht daher folgendermaßen aus (Die Funktion main darf nur einmal pro ausführbarem Programm und nicht in jeder compilierten Datei vorhanden sein):

Preprocessor Anweisungen (#define, #include, ...)

externe Variablendefinitionen
static externe Variablendefinitionen
extern - Verweise auf anderswo definierte externe Variable
Prototypen

Funktion main

Funktion 1
.
.
.
[Funktion n]

Eine Funktion ist dabei folgendermaßen aufgebaut:

[speicherklasse-f] [datentyp-f] funktionsname ([formale Parameter])

[Deklaration formale Parameter;] /* nur im alten K&R - C, */
/* im Ansi C ist die Deklaration */
/* direkt in den runden Klammern möglich */

{
[Definition lokaler Variablen;]

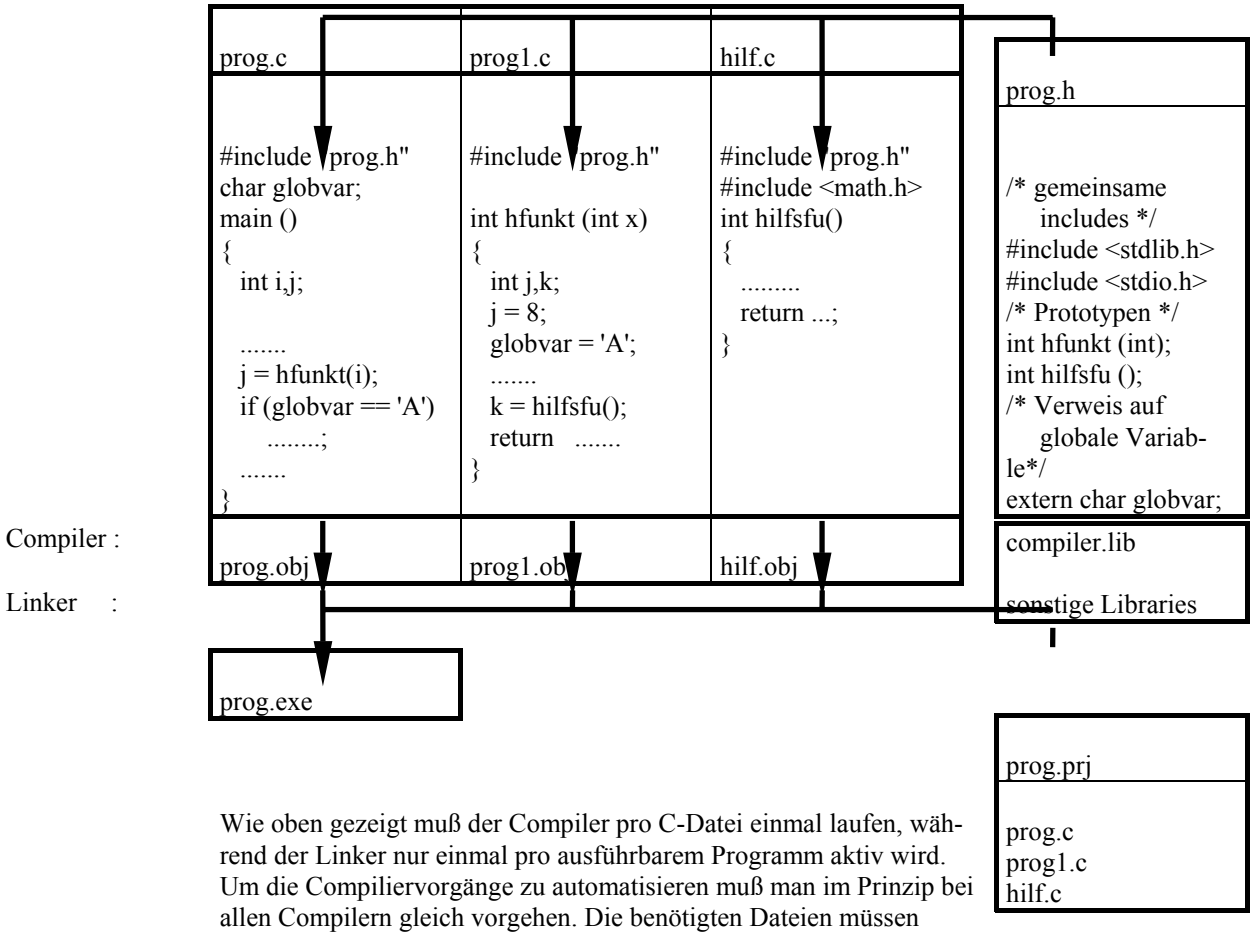
[Anweisungen;]

}

```
Es wäre die folgende Situation denkbar:

#include "stdio.h"  /* notwendig, wenn I/O Funktionen verwendet werden   */
#define EOF   -1    /* wenn im Prog EOF vorkommt wird es durch -1 ersetzt*/

int var_ext_int=8; /* diese Variable ist extern, sie ist in den gemeinsam*/
                   /* compilierten Funktionen bekannt und kann auch in  */
              /* anderen (gesondert compilierten) Funktionen bekannt sein */

static int var_ext_dat;    /* diese Variable ist in allen Funktionen dieser*/
                           /* Datei bekannt (gemeinsam compilierte Funkt. )*/

extern int sonst_var; /* Verweis auf eine, in einer anderen (getrennt
                         compilierten) Funktion definierten externen
                         Variablen; wenn extern angegeben ist,
                         so wird kein Speicherplatz belegt !         */

float funkt_privat (char [],int);  /* Prototyp für eine
                                      (selbstgeschriebene) Funktion */

main ()                            /* Hauptfunktion, hier startet
                                      das Programm */
{
   int i=0; float f;       /* lokale Variable, nur in main bekannt    */
   static int i_stat;      /* lokale statische Variable               */
      ......................
      f = funkt_privat ("ein String",i) - 2;   /* das ist der Aufruf der
                                                  Funktion funk_privat */
      ......................
}

float funkt_privat(char instring[],int k) /* Definition der Funktion
                                             funkt_privat            */
{
   float ff;
   ......................
   ......................
   return ff;             /* korrekte Rückkehr mit einem float Wert  */
}
```

Bei sinnvoller Modularisierung hat man schon bei mittleren Programmen relativ viele Funktionen. Man legt für ein Programm ein Subdirectory an, wo für jede Funktion eine Datei (funkname.c) existiert. Bei zuvielen Funktionen kann man auch logisch zusammengehörige Funktionen gemeinsam in einer Datei (gruppe.c) unterbringen. Als Verbindung zwischen all diesen Dateien fungiert die programmeigene Header Datei (progname.h). Wird eine Funktion geändert, dann wird nur diese eine Datei compiliert und nachher werden alle Objekt Files neu zum progname.exe gelinkt. Bei allen C Compilern kann man die Compilier und Linkvorgänge mittels Make bzw. Projektdateien vollständig automatisieren.

Beispielsweise könnten folgendende Dateien existieren :



Wie oben gezeigt muß der Compiler pro C-Datei einmal laufen, während der Linker nur einmal pro ausführbarem Programm aktiv wird. Um die Compiliervorgänge zu automatisieren muß man im Prinzip bei allen Compilern gleich vorgehen. Die benötigten Dateien müssen namentlich im Projekt-File genannt werden. Im Turbo-C 2.0 mußte man noch eine normale Datei namens prog.prj anlegen, in der die Dateinamen aller für das endgültige C Programm notwendigen Dateien stehen mußten. In allen neueren Compilern gibt es eigene Menüpunkte (Open Project), wo man die Zusammengehörigkeit mehrerer Dateien zu einem Programm festlegen kann. Es wäre noch möglich zusätzliche xx.c oder xx.obj oder xx.lib Dateien anzugeben, falls benötigt. Die compilerspezifische name.lib Datei wird automatisch von Linker verwendet. Eine xx.lib Datei enthält lediglich mehrere xxx.obj Dateien, die vom Linker verwendet werden falls dies notwendig ist.

7.4. lokale / globale Variable (Speicherklasse, Lebensdauer und Gültigkeitsbereich)

Allgemein gesehen gibt es in guten Programmiersprachen globale und lokale Variable.

Variable ist	C-Definition ist	Lebensdauer	Gültigkeit	static Zusatz
lokal	in einer Funktion	nur während die Funktion läuft	nur im Programmblock	Lebensdauer ist Programmlaufzeit
global	außerhalb einer Funktion	die gesamte Programmlaufzeit	gesamtes Programm	Gültigkeit ist nur mehr Datei

Syntax bei der Variablendefinition :

[Speicherklassenattribut] Datentyp [Typattribut] Name[=Initwert][,Name[=Initwert]]...;

Die Speicherklasse einer Variablen (lokal oder global) wird nur durch den Ort der Variablendefinition bestimmt. Die Speicherplatzattribute beeinflußen die Eigensschaften der Variable (static), informieren den dem Compiler (extern, register) bzw. oder dienen überhaupt anderen Zwecken (typedef).

lokale Variable (defniert innerhalb der {} einer Funktion). Diese Variable ist nur innerhalb der Funktion (bzw. des Programmblocks) gültig. Bei jedem Aufruf der jeweiligen Funktion wird die Variable neu angelegt, beim Funktionsende verworfen.

globale Variable (defniert außerhalb der {} einer Funktion). Variable, die außerhalb von Funktionen definiert werden, sind globale Variable, sie leben während des ganzen Programms und sind überall im Programm bekannt. Damit der Compiler solche Variable auch in gesondert compilierten Funktionen korrekt erkennt, ist dort eine Deklaration notwendig, die genauso aussieht wie die Definition, allerdings schreibt man das Speicherplatzattribut „extern“ davor

```
int varname;            /* globale Variable, definiert außerhalb von {} */

extern int varname;     /* Deklaration, notwendig in anderen Dateien, die zum
                           gleichen Programm gehören, informiert den Compiler,
                           daß es eine globale integer Variable namens varname gibt.*/
```

static als Zusatz zu lokalen Variablen verändert die Lebensdauer; diese ist jetzt das gesamte Programm. Der Gültigkeitsbereich, bei lokalen in der Funktion definierten Variablen bleibt aber die Funktion. Wird static bei externen Variablen (außerhalb der Funktion) verwendet, so ändert sich die Gültigkeit - die Variable ist nur mehr begrenzt global - sie ist nur mehr in den Funktionen der Datei bekannt, in der sie definiert wurde. extern - Deklarationen sind nicht möglich.

register ist nur bei lokalen Variablen möglich; diese Angabe sagt dem Compiler, daß er die Variable in ein Register legen soll (aber nicht muß). Deshalb kann der Operator & (Adresse von) nicht auf Variablen mit dem Speicherplatzattribut register angewendet werden. Register ist eigentlich keine richtiges Speicherklassenattribut und bei den heutigen Compilern auch nicht mehr notwendig.

typedef hat eigentlich nichts mit Speicherklassen zu tun, bietet auch nicht die Möglichkeit neue Typen zu vereinbaren, man kann lediglich neue (synonyme) Namen für bestehende Typen vergeben. Üblicherweise werden die durch typedef erzeugten Namen in Großbuchstaben geschrieben.
Der bekannteste typedef ist "typedef struct _iobuf FILE". Man kann daher jederzeit FILE schreiben, obwohl immer eine Struktur vom Typ _iobuf gemeint ist.

Speicherklasse	Gültigkeit	Lebensdauer
global	gesamtes Programm	gesamtes Programm
lokal	Block (Funktion)	Block (Funktion)
register	Block (Funktion)	Block (Funktion)
static (bei lokal)	Block (Funktion)	gesamtes Programm
static (bei global)	Datei	gesamtes Programm

Bei Funktionen gilt normal global als Speicherklasse. (weil die Funktion außerhalb anderer Funktionen definiert wird). Static ist als Zusatz möglich, wodurch die Funktion nur mehr von (anderen) Funktionen derselben Datei aufgerufen werden kann.

Hier folgt ein an sich schlechtes Programm, der Sachverhalt wird aber gut demonstriert

```
int count=33;     /* das ist eine globale (externe) Variable */
static int zaehl; /* das ist eine beschränkt globale, nur innerhalb
                     der Datei (des gemeinsam compilierten Codes)
                     gültige externe Variable   */
main()
{
register int index; /* lokale (auto) Variable, nur in main bekannt */

   head1();        /* die erste der 3 Funktionen wird aufgerufen */
   head2();       /* bei 1 und 2 werden die Rückgabewerte ignoriert */
   index = head3();
   printf("in main count (extern) = %d\n",count);
   for (index = 8;index > 0;index--) {
      int stuff=0; /* lokale (auto) Variable, nur gültig bis zur }     */
                   /* Gültigkeitsbereich nur im Block {}, wo definiert */
         stuff++;
         printf("stuff = %d ",stuff);
      printf(", index is now %d\n",index);
    }
}

int counter;      /* globale (externe) Variable, gültig von hier an     */
                  /* Jede Variable gilt ab dem Punkt, an dem sie
                     definiert wurde (im jeweiligen Gültigkeitsbereich) */
head1()
{
int index;        /* lokale (auto) Variable, nur in head1 bekannt */

   index = 23;
   printf("in head1, index = %d\n",index);
     /* mangels return endet die Funktion am Blockende, der
        Rückgabewert (per default vom Typ int) ist undefiniert  */
}

head2()
{
int count; /* dies hier ist eine nur in head2 bekannte lokale Variable,  */
           /* sie ersetzt innerhalb von head2 die globale Variable count */
           /* trotzdem sollten Variable gleichen Namens nie vorkommen    */

   count = 53;
   printf("in head2 count (lokal) = %d\n",count);
   counter = 77;
}
head3()
{
   printf("in head3, counter = %d\n",counter);
   return (5);    /* das ist eine korrekte, wenn auch immer gleiche
                     Funktionswertrückgabe */
}
```

8. OPERATOREN in C (davon gibt es viele)

Rang	Operator	Bemerkung		Auswertung
1	**()**	bei Funktionsaufruf, Klammerung		li Õ re
	[]	normale Tabellenindizierung		
	.	zum Ansprechen von Strukturelementen		
	->	Strukturelemente mittels Pointer auf Struktur		
2	**(cast)**	unär erzwungene Datentypkonvertierung		re Õ li
	*****	unär Inhalt von (nur bei Pointer)		
	&	unär Adresse von		
	-	unär das minus Vorzeichen		
	+	unär das plus Vorzeichen		
	!	unär logische Negation (NOT)		
	~	unär bitweises Komplement		
	++	unär Inkrement (x = x + 1)		
	--	unär Dekrement (x = x - 1)		
	sizeof	unär Speicherplatz in Byte z.B. sizeof(x)		
3	*****	Multiplikation		li Õ re
	/	Division		
	%	Modulo (Restdivision)		
4	+	Addition		li Õ re
	-	Subtraktion		
5	>>	Rechts Shift (bitweise)		li Õ re
	<<	Links Shift (bitweise)		
6	<	kleiner Vergleichsoperator		li Õ re
	>	größer Vergleich		
	<=	kleiner gleich Vergleich		
	>=	größer gleich Vergleich		
7	==	Vergleich auf Gleichheit		li Õ re
	!=	Vergleich auf Ungleichheit		
8	**&**	bitweises und		li Õ re
9	^	bitweises exclusiv oder (EOR)		li Õ re
10	|	bitweises oder		li Õ re
11	**&&**	logisches und	strikt	li Õ re
12	||	logisches oder	strikt	li Õ re
13	**?:**	bedingte Bewertung	strikt	re Õ li
14	=	Zuweisung		re Õ li
	+=	verkürzte Zuweisung (x += 1)		
15	**,**	Kommaoperator zur Statementtrennung	strikt	li Õ re

C ist bekannt für eine Unzahl von Operatoren. Alle Operatoren können theoretisch in einem einzigem Ausdruck vorkommen, dessen Ergebnis beispielsweise einer Variablen zugewiesen wird oder aber auch als Ergebnis einer Bedingung in einem if Statement verwendet wird.

Bei vielen Operatoren gibt es auch eine aufwendigere Einteilung in Ränge (mit Prioritätsreihenfolge) und eine Auswertereihenfolge innerhalb des Rangs.

Der Rang bestimmt die Prioritätsstufe; Hoher Rang (1,2) bedeutet hohe Priorität. Operatoren am gleichen Rang haben die gleiche Priorität. Sie werden gemeinsam betrachtet und gemäß der Auswertereihenfolge (von links nach rechts oder von rechts nach links) abgearbeitet, je nachdem wo sie im Ausdruck vorkommen.

Die meisten Operatoren sind auch in C binäre Operatoren mit infix Schreibweise, die als Operand Operator Operand geschrieben werden. Die unären Operatoren werden größtenteils in prefix Notation (Operator Operand) verwendet, einige (++, --) sind auch postfix (Operand Operator) einsetzbar.

	unär	binär
infix		**a + b**
prefix	**++a**	+ a b
postfix	**a++**	a b +

Es ist im Gegensatz zu anderen Sprachen nicht einfach bei komplexen Ausdrücken den Überblick zu behalten. Klammerung kann immer eingesetzt werden um die normale Prioritätenfolge zu durchbrechen und damit auch die Ausdrücke übersichtlicher zu machen. Machen Sie in komplexen Ausdrücken Klammern auch dort, wo sie aufgrund der Prioritätsreihe gar nicht notwendig wären; damit wird der Code leichter lesbar.

Beispielsweise sind ++(*++(*p))->trp.s oder ++a&&++b||*c durchaus korrekte, wenn auch schwer verständliche Ausdrücke.

Weil es hier einige logische Operatoren gibt, muß noch einiges zur Abarbeitung derselbigen gesagt werden. Während man in anderen Sprachen an einer Stelle, wo eine Bedingung erwartet wird, keinen beliebigen Ausdruck schreiben kann, ist dies in C durchaus möglich.

if (a + 3) .... ist erlaubt und sogar sinnvoll einsetzbar. Jeder Ausdruck bringt ein Ergebnis, das in einem Datentyp vorliegt, der sich (mittels Konvertierung) aus den Datentypen der einzelnen Operanden ableiten läßt. Wenn dieses Ergebnis für eine Bedingung genutzt wird, dann wird das Ergebnis des Ausdrucks (bzw. der Inhalt einer Variablen) verwendet um die Bedingung mit WAHR oder FALSCH zu bewerten. Der Wert 0 gilt als FALSE, alle anderen Werte als TRUE. Das Ergebnis von Vergleichsoperatoren (z.B. a > 0) ist ein int - Typ mit dem Wert 0 oder 1.

Es sind daher die folgenden Alternativen gleichwertig.

```
if (a > 5) ....          v=a > 5;    if (v) ....
if (b == 2) ...          w=b == 2; if (w) ....
```

Jede Variable kann eine logische Variable sein

Kurz zusammengefasst gilt also:

Jeder Datentyp ist als logische Variable verwendbar. FALSE=0, TRUE != 0

Wichtig

Weil das Ergebnis aller Ausdrücke bzw. der Inhalt aller Variablen als Zahl betrachtet werden kann, sind im Programmcode auch beliebige Ausdrücke erlaubt, wo eine Bedingung ausgewertet wird.

Wenn als Wert 0 herauskommt, dann false, bei allen anderen Werten true; (3 > 4) liefert also 0, (5 < 7) liefert 1 oder besser ungleich 0. Steht aber beispielsweise -23 in einer int Variablen, so gilt das auch als true. Verschiedene C Funktionen liefern irgendwelche Werte ungleich 0 wenn sie TRUE liefern.

() Die runde Klammer wird benötigt, um die aufgrund der Prioritätsregeln der Operatoren gegebene Abarbeitungsreihenfolge in Ausdrücken zu verändern z.B. (a+b)*c. Bei Funktionen und Funktionsaufrufen stehen die Parameter auch in runden Klammern.

[] Für die Definition von Tabellen bzw. Arrays und auch für das Ansprechen einzelner Tabellenelemente im Programmablauf einsetzbar.

. Beim normalen Ansprechen von Elementen einer Struktur muß immer zuerst der Strukturname, dann der Punkt und dann der Elementname angeführt werden; also (strukt.unterstrukt.element). Auch in numerischen Konstanten findet der Punkt als Dezimalpunkt bei Gleitkomma Verwendung.

-> Verfügt man über einen Pointer, der auf eine Struktur zeigt (die Pointervariable enthält die Adresse der Struktur), dann ist nicht mehr der Punkt zu verwenden, sondern der Operator ->. Ein normaler Zugriff wäre folgendermaßen zu formulieren: ptr_struktur->element.

(cast) Damit können beliebige Konvertierungen innerhalb eines Ausdrucks durchgeführt werden. An die Stelle von cast ist jeweils der Datentyp zu schreiben, nach dem konvertiert werden soll. Konvertiert wird der Ausdruckteil (dessen Ergebnis), der nach (cast) steht. Wenn man sauber programmieren will, dann ist es günstig, unterschiedliche Typen nicht zu vermischen, sondern wenn nötig, eine Konvertierung mit dem cast Operator durchzuführen, auch wenn die gleiche Konvertierung automatisch durchgeführt würde. Beispielsweise:

```
float f;
int i;      i = (int) f;      f = (float) i;
```

***** Dieser Operator kann nur auf Pointer angewendet werden, Es wird der Inhalt des Feldes bereitgestellt, auf das der Pointer gerade zeigt. Weil Pointer in C an einen Datentyp gebunden sind, wird also ein Feld (dessen Inhalt) in diesem Datentyp bereitgestellt.

& Will man einem Pointer einen Wert zuweisen, so ist es sinnvoll, die Adresse einer Variablen zu verwenden. Mit diesem Operator wird die Adresse eruiert. Im anschliessenden Kapitel werden & und * Operator näher erklärt.

- + Das Vorzeichen für negative numerische Werte. Beispielsweise -222 -x -(x+y). Es gibt in Ansi C auch ein + als unäres positives Vorzeichen.

! Als logische Negation wird das ! verwendet. Als logischer Operator liefert das ! generell nur die int Ergebniswerte 0 oder 1.

```
x =5;          !x        liefert 0
x=6;y=6;     !(x-y)     liefert 1
```

~ Das Zeichen ~ (die Tilde) erzeugt das bitweise Komplement der jeweiligen Variablen, die aber nur int oder char sein darf.

++ Der Inkrementoperator (erhöht Wert um 1) ist in der in C vorkommenden Form eine Spezialität. Er erhöht den Operanden selbst (!!! sonst wird der Operand nie verändert) um 1 und gibt abhängig von der postfix oder prefix Schreibweise den Ursprungswert oder den um 1 erhöhten Wert als Ergebnis weiter. Daher kann ++i auch verwendet werden, ohne eine Zuweisung zu tätigen. ++ kann praktisch nur auf Variable und nicht auf Ausdrücke angewandt werden. Beispiele:

```
int i,x=2,y=3,z=4;
i =x++;                 /* Wert danach x=3, i=2            */
y++;                    /*          y=4                    */
i = ++z;                /*          i=5, z=5               */
printf("- %d -",++i);   /*          i=6, gedruckt wird 6   */
x=3;y=4;i=++x + y++;    /*  i=4+4     i=8, x=4, y=5        */
i = ---x;               /*  Syntaxfehler (-- wird auf den Ausdruck -x angewendet)
                            ++ und -- dürfen aber nur auf sogenannte L-Werte ange-
                            wendet werden. Das sind solche, die links von einer
                            Zuweisung stehen dürfen  */
i = --x - x * x--;      /*  Sollten vermieden werden, das Ergebnis ist nicht
                            eindeutig, es gibt Compiler-Unterschiede. Es bleibt
                            dem Compiler überlassen zuerst den Teil "--x"
                            auszurechnen oder rechts zu beginnen und vorher
                            "x * x--" zu ermitteln. Auch Statements wie
                            a[i] = ++i; sollte man tunlichst unterlassen ! */
```

Wichtig

Wenn in einem Ausdruck (ganze Zuweiseung, etc.) eine Variable zweimal vorkommt, so darf man auf sie nicht ++ oder -- anwenden !

-- Dekrementierung (Wert um 1 vermindern) äquivalent zu ++ einsetzbar

sizeof Wenn man dynamisch während der Laufzeit Hauptspeicherplatz anfordert, dann will man in der Regel Platz für bestimmte Variablen. Es ist dann sehr günstig sizeof einzusetzen, weil damit der benötigte Speicherplatz für bestimmte Datentypen (aber auch Strukturen) ermittelt werden kann.

sizeof(int) liefert 2 oder 4 als Ergebnis. sizeof (stru_datum) würde also 6 oder 12 liefern.

***** Als binärer Operator ist * die normale Multiplikation x= y * z;

/ Damit wird die übliche Division durchgeführt. Sind beide Operanden int so ist auch das Ergebnis int (Nachkommastellen werden abgeschnitten). Ist aber ein Operand float, bzw. wird ein Operand mit dem cast Operator nach float konvertiert, so gibt es float als Ergebnistyp. z = x/y liefert int, z = (float) x / (float) y liefert float.

% Modulo (der Rest einer Ganzzahldivision) wird mit dem % ermittelt. Es ergibt also 23%7 den Wert 2. Wenn x und y beide vom Typ int sind, dann gilt (x/y)*y + x%y == x.

+ Selbstverständlich kann auch addiert werden.

- Als binärer Operator bewirkt - die Subtraktion.

>> Das ist ein Spezialoperator für den bitweisen Rechtsshift.

5 >> 2 bedeutet, daß in der internen Darstellung von 5 alle Bits um zwei Stellen nach rechts verschoben werden. Die rechten beiden Bits gehen dabei verloren, links wird mit 0-Bits aufgefüllt. Das Ergebnis ist daher 1. Bei negativen Zahlen sind shifts problematisch, günstig wäre es, wenn nicht mit 0, sondern mit 1 aufgefüllt würde; das machen aber nicht alle Compiler. Der Operator ist nur auf int anwendbar, die Anzahl Bits muß ebenfalls einen sinnvollen Wert haben. 9 hat intern die Bitkombination 00..01001, 9>>2 ergibt also 00..00010 das Ergebnis ist daher 2.

<< Das ist der Linksshift, der sinngemäß gleich funktioniert.
x=1; x = x << 2; printf("- %d -",x); ergibt 4.

> < <= >= == != sind die Vergleichsoperatoren. Diese funktionieren wie üblich, es ist aber wichtig den Gleichheitsoperator == nicht mit dem Zuweisungsoperator = zu verwechseln. Ansonsten bleibt die Anweisung syntaktisch korrekt, es wird aber eine Variable unkon-

trolliert mit einem Wert versorgt und die Bedingung wird immer true (oder auch immer false) ausgehen. Generell liefern die Vergleichsoperatoren einen int Datentyp, der eine Zahl ungleich 0 (meist 1) für true oder 0 für false enthält. Dieses Ergebnis entspricht der schon oben erwähnten logischen Variablen.

& Will man zwei Operanden vom Typ integer bitweise nach der AND-Regel verknüpfen, kann das mit & erreicht werden. 6 & 3 ergibt dabei 2 weil die Bitkombination 110 verknüpft mit 011 zwangsweise 010 als Ergebnis liefert. Sie sollten das bitweise und (&) nicht mit dem logischem und (&&) verwechseln, auch wenn das meist keine schlimmen Auswirkungen auf Ihr Programm haben wird (z.B. 110 & 001).

^ Auch exclusiv oder kann mittels ^ bei der Bitverknüpfung eingesetzt werden. 6 ^ 3 ergibt 5 (110 ^ 011 liefert 101).

| Da Sie das bitweise oder ebenfalls durchschauen ist völlig klar, daß der Ausdruck 6 | 3 als Ergebnis 7 hat.

&& Das logische und wird mit dem Operator && realisiert. Die Operanden sollten zwei logische Datentypen sein. Sind beide Operanden ungleich 0, dann ist das Ergebnis 1 sonst immer 0. 6 && 1 liefert als Ergebnis True (true && true ergibt true).

|| Als letzter logischer Operator wird mit || das oder dargestellt. Wie allgemein bekannt wird das Ergebnis false (0), wenn beide Operanden 0 sind, sonst gibt es immer das Ergebnis True. 6 || 3 ergibt natürlich auch True.

?: Fragezeichen Doppelpunkt ist ein spezieller C Operator, der zwar den Neuling verwirrt aber sehr effizient einsetzbar ist. Sinngemäß kann man damit einen if-Befehl mitten in einem Ausdruck unterbringen. Der Operator hat drei Operanden, ist also ternär.

if (a < b) x = 2 else x=3; gleichwertig ist die folgende Anweisung x = a<b ? 2 : 3;

Allgemein formuliert gilt: (bedingung) ? true-Teil : false-Teil.

Es ist aber zu beachten, daß true-Teil und false Teil einen Wert liefern müßen, der abhängig von der Bedingung das Ergebnis des gesamten Operators (der sogenannten bedingten Bewertung) liefert.

Zur Maximum-Bildung würde sich eignen: max = (a>b) ? a : b;
laienhafte Mathematiker könnten schreiben k = b == 0 ? 0 : a/b;

= Auch die Zuweisung gilt als Operator, aber mit geringer Priorität, damit auch der ganze Ausdruck berechnet wird, bevor die Zuweisung erfolgt. Achten Sie auf die Auswertungsrichtung (re Õ li).

+= Es gibt eine verkürzte Schreibweise für die Zuweisung, bei der ein wenig an Schreibarbeit eingespart wird. Anstelle einer üblichen Schreibweise x = x + 4; kann auch x += 4; geschrieben werden.

Generell kann statt x = x Operator Ausdruck
als kurze Variante auch x Operator= Ausdruck geschrieben werden.
Dies funktioniert mit den Operatoren + - * / % & ^ | << >>.

, Mit dem Komma können mehrere einzelne Ausdrücke zu einem einzigen zusammengefaßt werden. Das Ergebnis des letzten (rechtesten) Teilausdrucks ist dann das Ergebnis des gesamten Ausdrucks. Es ist auch möglich, anstelle der Blockung {} mit dem Komma mehrere Einzelausdrücke zusammengefaßt hinter eine while (oder eine andere) Anweisung zu stellen. Soll der Komma Operator innerhalb einer durch Beistrich getrennten Liste stehen (z.B. bei Funktionsaufrufen), so muß er in runde Klammern gestellt werden.

Und nun einige Beispiele:

Dieses Beispiel zeigt die strikte links - rechts Abarbeitung der Operatoren && und ||

Wichtig

```
#include <stdio.h>
void main()  /* Dieses Beispiel zeigt die (logischen) Vergleichsoperatoren
*/
{
   int x = 11,y = 11,z = 11;
   char a = 40,b = 40,c = 40;
   float r = 12.987,s = 12.987,t = 12.987;

   if (x == y) z = -13;   /* z wird auf -13 gesetzt */
   if (x > z)  a = 'A';   /* a wird auf 65  gesetzt (Ascii Code "A")  */
   if (!(x > z)) a = 'B'; /* nichts wird geändert   */
   if (b <= c) r = 0.0;   /* r wird auf 0.0 gesetzt */
   if (r != s) t = c/2;   /* t wird auf 20 gesetzt  */

   if (x = (r != s)) z = 1000; /* x erhält einen Wert ungleich 0
                                  und z wird auf 1000 gesetzt */
   if (x = y) z = 222;   /* dies setzt x = y, and z = 222 */
   if (x != 0) z = 333;  /* dies setzt z = 333 */
   if (x) z = 444;       /* dies setzt z = 444 */

   x = y = z = 77;
   if ((x == y) && (x == 77)) z = 33; /* gesetzt wird z = 33 */
   if ((x > y) || (z > 12))   z = 22; /* gesetzt wird z = 22 */
   if (x && y && z) z = 11;           /* gesetzt wird z = 11 */
   if ((x = 1) && (y = 2) && (z = 3)) r = 12.00; /* es wird zugewiesen:
                              x = 1, y = 2, z = 3, r = 12.00 */

   if ((x == 2) && (y = 3) && (z = 4)) r = 14.56;
                         /* In dieser Zeile wird keine
                            Zuweisung durchgeführt (trotz y=3 !!!)  */
                         /* Bei den Operatoren && || , ?: ist nämlich die
                            Abarbeitung von links nach rechts garantiert  */
                         /* Bei && und || wird der Ausdruck nicht mehr
                            weiter untersucht, wenn das Ergebnis feststeht*/

   if (x == z); z = 27.345;  /* z wird immer gesetzt  ";"!!!! */
   if (x != x)  z = 27.345;  /* nichts tut sich                        */
   if (x = 0)   z = 27.345;  /* dies setzt x = 0, z bleibt unverändert */

}
```

```
#include <stdio.h>
void main()
{
int x = 0,y = 2,z = 1025;
float a = 0.0,b = 3.14159,c = -37.234;


   x = x + 1;       /* Incrementierung */
   x++;
   ++x;
   z = y++;         /* z = 2, y = 3 */
   z = ++y;         /* z = 4, y = 4 */


   y = y - 1;       /* Decrementierung */
   y--;
   --y;
   y = 3;
   z = y--;         /* z = 3, y = 2 */
   z = --y;         /* z = 1, y = 1 */

                                  /* verkürzte Zuweisungen */
   a = a + 12;      /* 12 zu a addieren */
   a += 12;         /* nochmal 12 zu a addieren */
   a *= 3.2;        /* a mit 3.2 multiplizieren */
   a -= b;          /* b von a abziehen */
   a /= 10.0;       /* a durch 10 dividieren */
```

```
                                    /* bedingte Bewertung */
   a = (b >= 3.0 ? 2.0 : 10.5 );    /* Dieser Ausdruck    */

   if (b >= 3.0)                    /* und dieses         */
      a = 2.0;                      /* Statement          */
   else                             /* sind im Resultat   */
      a = 10.5;                     /* gleich             */

   c = (a > b?a:b);        /* auch als max */
   c = (a > b?b:a);        /* oder min einsetzbar */
}
```

In der C - Bibliothek stdlib.h ist die Funktion atoi enthalten. Damit können Zahlen, die lesbar (als Zeichen) in einem String stehen in eine integer Zahl umgewandelt werden. Auch ein scanf mit einer %d Formatierung arbeitet mit der selben Logik. Welche Vor- und Nachteile diese Funktion hat sieht man beim Studium des Codes.

```
int myatoi (char s[]) /* einen String nach integer umwandeln           */
                /* der Parameter, ein String (char-Arry), endet mit \0 */
{
   int i,n,vorzeichen;
   for (i = 0; s[i]==' ' || s[i] == '\n' || s[i] == '\t';i++) ;
               /* überlesen von Zwischenräumen (blanks,newline,Tab) */
   vorzeichen = 1;
   if (s[i] == '+' || s[i] == '-')      /* ermittle Vorzeichen wenn  */
          vorzeichen = (s[i++] == '+') ? 1 : -1;  /* vorhanden     */
   for (n = 0; s[i] >= '0' && s[i] <= '9';i++)    /* errechnen des */
          n = 10 * n + (s[i] - '0');    /* Wertes im integer - Feld  */
   return (vorzeichen * n);             /* aus den Einzelstellen     */
}
```

9. Dateien in C

9.1. Grundlegendes zu Dateien

Daten sind Aussagen über Persone, Gegenstände, Sachverhalte in Form von Zahlen, Text oder Bildern, die Information über diese Personen, Gegenstände oder Sachverhalte jemandem dritten liefern können

Beispiele :

Person	Gegenstand	Sachverhalt
M. Maier	Opel Kadett	Aktenzahl U44/95
geb. 3.12.68	Bj 88	Müller gegen Huber
A-Gasse 43	74000 km	Streitwert 100 000,-
1234 A-dorf	VB 60000	

Es fehlen möglicherweise Angaben wie

Geschlecht	Farbe	wer sind Müller oder Huber
Ehestand	Erstzulassung	Worum wird gestritten
Krankenkasse	Sonderausstattung	bei welchem Gericht

Jedes Datenelement(Datum) liefert eine Aussage. Manche wie VB 60000 enthalten auch versteckte bzw. mehrere Aussagen (Preis und Verhandlungsbereitschaft über diesen).

WICHTIG: Es kommt auf den Blickwinkel an, unter dem die Daten gesammelt werden. Ein Autoverkäufer wird andere Daten über Fahrzeuge sammeln als ein Händler oder die Polizei; ein Arzt andere Personendaten als ein Personalbüro oder eine Heiratsvermittlung.

Strukturieren der Daten am Beispiel eines Autos :

Ein Privater möchte einen Gebrauchtwagen kaufen und findet folgende Annonce: Opel Kadett B, Bj 1988, 74000 km, Pickerl bis 12/95, Schiebedach, Vb 60000

Um mehrere Angebote leicht vergleichen zu können, legt er sich eine Tabelle an:

Marke	**Type**	**Baujahr**	**gefahrene km**	**Pickerl**	**Extras**	**Preis**
Opel	Kadett B	1988	74000	12/95	Schiebedach	60000
Opel	Kadett	1992	61000	8/96	Radio Alufelgen Anhängekupplung	75000
VW	Golf	1991	36500	2/96	-	67000
Ford	Escort	1978	1022000	-	-	5000

Datensätze mit Datenfelder

Namen der Datenfelder

Beim erfassen der Daten fällt folgendes auf:

- die Information VB beim Preis ging verloren (weil man nur Zahlen angeben wollte)
- es ist unklar wieviele Extras ein Auto haben kann

Wird diese Tabelle nicht auf Papier sondern in einem Computer (auf der Festplatte) gespeichert, so werden die Daten in einer Datei (engl. File) gespeichert. Ein Spalte (ein Kästchen) aus der Tabelle nennt man Datenfeld (oder Datenelement), eine Zeile heißt Datensatz.

Jedes Datenfeld hat einen Namen und entspricht einer Variablen.

- Datenfelder sind Variable.
- Mehrere logisch zusammengehörige Datenfelder finden sich in einem Datensatz
- Mehrere gespeicherte Datensätze bilden eine Datei.

Sinnvollerweise wird man sich ein Verzeichnis der Datenfelder eines Datensatzes anlegen.

Feldname	Feldart	Länge	Wertebereich	Bdeutung
Marke	Text	20	alphanum	wie im
Type	Text	20	alphanum	Typenschein
Baujahr	numerisch	4	1900-2010	Baujahr
gefahrene km	numerisch	6	0 bis 999999	Laufleistung
Pickerl	numerisch	4	100 bis 10000	Pickerl bis
Extras	Text	30	alphanum	Extras
Preis	numerisch	10	0 bis 9999999	Preis
Preisart	Text	2	FP, VB	Preisart

In Programmiersprachen nennt man solch eine Zusammenfasung von einzenen Variablen unterschiedlichen Typs eine Struktur.

9.2. Strukturen (Zusammenfassen von Elementen unterschiedlichen Typs)

Strukturen gliedern mehrere logisch zusammengehörige Datenfelder zu einer Einheit. Das folgende Beispiel definiert ein Struktur direkt.

```
struct { int tag;            /* definieren einer Struktur datum1, mit */
         int monat;          /* den 3 Elementen tag,monat,jahr        */
         int jahr; } datum1;
```

Will man an verschiedenen Programmstellen eine Struktur gleichen Typs nutzen, so ist die folgende Variante zu empfehlen.

```
struct stru_datum { int tag;         /* Deklaration einer Struktur vom   */
                    int monat;       /* Typ stru_datum.                  */
                    int jahr; };     /* !!! belegt keinen Speicherplatz  */

struct stru_datum dat1,dat2;         /* Definition der Strukturen dat1,  */
                                     /* dat2  (vom Typ stru_datum)       */

struct stru_datum dat3 = {27,2,1997};/* Initialisierung der 3 Felder     */
dat1.tag = 14; dat1.monat=9;     /* einzelne Strukturelemente ansprechen */
dat2 = dat1;                     /* ganze Struktur ansprechen            */
                                     /* Beispiel mit Unterstruktur */

struct stru_angestellter { char name [20];
                           char vorname [20];
                           struct stru_datum geburtsdatum; };

struct stru_angestellter personal;   /* die Definition der Struktur      */

personal.geburtsdatum.tag = 24;      /* befüllen einzelner Elemente      */
personal.name[0] = 'H';
strcpy(personal.vorname,"Wilma");
```

Weil Strukturen mehrere Elemente unterschiedlichen Typs enthalten müssen auch alle einzelnen Elemente mit Namen und Typ in geschwungenen Klammern angeführt werden. Als Elemente sind alle skalaren Datentypen, aber auch Arrays, andere Strukturen und Pointer zugelassen. Es ist sinnvoll, den Aufbau einer Struktur gesondert zu deklarieren und bei der Definition die Elemente nicht mehr aufzuzählen.

Die Elemente von Strukturen werden mit dem . Operator (als strukturname.elementname) angesprochen. Wenn man Pointer auf Strukturen verwendet, dann wird der Operator `->` (als pointer->elementname) eingesetzt. Neben diesen beiden Operatoren kann man auf Strukturen nur mehr den Operator & anwenden. Dieser liefert die Adresse der Struktur (den bloßen Strukturnamen zu verwenden ist nur möglich, wenn man eine Struktur einer anderen zuweist).

```
#include <stdio.h>
void main()
{
struct person {
   char initial;    /* erster Buchstabe des Zunamens */
   int age;         /* Alter                         */
   int grade;       /* irgendeine Punktezahl         */
   } ;

struct person boy,girl;

   boy.initial = 'R';
   boy.age = 15;
   boy.grade = 75;
   girl.age = boy.age - 1;  /* sie ist ein Jahr jünger */
   girl.grade = 82;
   girl.initial = 'H';
   printf("%c ist %d Jahre alt und hat eine Punktezahl von %d\n",
           girl.initial, girl.age, girl.grade);

   printf("%c ist %d Jahre alt und hat eine Punktezahl von %d\n",
           boy.initial, boy.age, boy.grade);
}
```

Kombination Array und Struktur

```
#include <stdio.h>
void main()
{
struct person {
   char name[20];   /* hier gibts den ganzen Namen */
   int age;
   int grade;
   };

struct person kids[12];

int index;

   for (index = 0;index < 12;index++) {
      strcpy (kids[index].name,".-Kind");
      kids[index].name[0] = 'A' + index;
      kids[index].age = 16;
      kids[index].grade = 84;
   }
   kids[3].age = kids[5].age = 17;
   kids[2].grade = kids[6].grade = 92;
   kids[4].grade = 57;

   for (index = 0;index < 12;index++)
      printf("%s ist %d Jahre alt und hat eine Punktezahl von %d\n",
             kids[index].name, kids[index].age,
             kids[index].grade);
}
```

```
         /* Dies ist auch eine typische Form der Strukturdeklaration,*/
         /* bei der nur der Aufbau der Struktur festgelegt wird.     */
struct tm {             /* Struktur für die Zeitfunktionen der time.h */
    int tm_sec;         /* seconds after the minute - [0,59] */
```

```
    int tm_min;         /* minutes after the hour - [0,59] */
    int tm_hour;        /* hours since midnight - [0,23] */
    int tm_mday;        /* day of the month - [1,31] */
    int tm_mon;         /* months since January - [0,11] */
    int tm_year;        /* years since 1900 */
    int tm_wday;        /* days since Sunday - [0,6] */
    int tm_yday;        /* days since January 1 - [0,365] */
    int tm_isdst;       /* daylight savings time flag */
    };
   /* Obige Definition steht in der Include Datei time.h.                */
   /* In ihrem Programm schreiben Sie dann:                              */
#include <time.h>
struct tm zeit_privat;
   /* Damit ist zeit_privat eine sogenannte unvollständige Struktur-     */
   /* definition, weil die Elemente bereits früher definiert wurden.     */
   /* Man sollte immer mit unvollständigen Strukturdefinitionen          */
   /* arbeiten, weil dadurch gleiche Strukturen (vom Typ struct tm)
      garantiert gleich sind. */
```

verschachtelte Strukturen

```
main()                    /* Strukturen können verschachtelt sein   */
{
struct person {
   char name[25];
   int age;
   char status;           /* V = verheiratet, S = single */
} ;

struct alldat {
   int grade;
   struct person descrip;
   char lunch[25];
};

struct alldat teacher,sub,student[53];

teacher.grade = 94;
teacher.descrip.age = 34;
teacher.descrip.status = 'V';
strcpy(teacher.descrip.name,"Hanil Sabal");
strcpy(teacher.lunch,"Döner Sandwich");

sub.descrip.age = 87;
sub.descrip.status = 'V';
strcpy(sub.descrip.name,"Old Lady Brown");
sub.grade = 73;
strcpy(sub.lunch,"Jogurt und Toast");
student[1].descrip.age = 15;
student[1].descrip.status = 'S';
strcpy(student[1].descrip.name,"Willy Binkel");
strcpy(student[1].lunch,"Teebutter");
student[1].grade = 77;

student[7].descrip.age = 14;
student[12].grade = 87;

}
```

9.3. Funktionen für die Dateibearbeitung mit Beispielen

Aus der Sicht eines Programms sind Dateien in der Regel Files , die in einem vom Betriebssystem unterstütztem Filesystem auf Festplatten oder anderen Datenträgern gespei-

chert sind. Bei vielen Betriebssystemen werden aber auch Geräte wie Tastatur, Druckerschnittstelle, serielle Schnittstelle, Bildschirm als spezielle Formen von Dateien betrachtet.

C sieht Dateien grundsätzlich als Streams (Datenströme - sequentiell hintereinander angeordnete Bytes mit Zeichen oder Binärdaten)..Jedes File ist eine geordnete Folge von Zeichen. Innerhalb der Zeichen gibt es bloß eine logische Gliederung :

9.3.1. Textfiles

bestehen aus ASCII-Zeichen, genormten Steuerzeichen(CR, LF, HT, VT,...) und <STRG> Z zum Markieren des Dateiendes. Ihre Programmdateien (name.c) sind typische Textdateien, sie haben eine Zeilenstruktur, wobei die Zeilen unterschiedlich lang und durch ein Zeilenende-Zeichen (CRLF) abgeschlossen sind.

Bei der I/O solcher Dateien werden durch scanf/printf numerische Daten von einer Zeichenfolge (in der Datei) in die internen Darstellung eines integers und umgekehrt konvertiert. Am MS-Dos System wird beim Einlesen aus CR,LF --> LF (\n), bei Ausgabeoperationen umgekehrt aus LF --> CR,LF.

Textdateien sind portabel (unter Umständen ist eine Zeichenkonvertierung notwendig, abhängig vom verwendeten Code).

9.3.2. binäre Files

haben keine allgemeingültige Struktur, sie entstehen als 1:1-Kopien von Pufferinhalten bei Schreiboperationen . Wird der Inhalt einer integer oder anderen Variablen in eine Datei geschrieben, so landet er unverändert Bit für Bit auf der Platte.

Zu binären Files benötigt man die zugehörige Dokumentation über ihren Aufbau, dann kann man die passenden Lesebefehle implementieren um die Daten wieder sinnvoll zu lesen.

Alle grafischen Dateiformen (.BMP, .GIF, .PCX, ...) oder auch der von Comiler und Linker erzeugte .OBJ und .EXE File sind binäre Dateien. Sie können im Handbuch der Grafikformate den Aufbau einer .BMP Datei studieren und diese z.B. komprimieren oder modifizieren. Das Betriebssystem kennt exakt den Aufbau einer .EXE Datei, was dringlich notwendig ist um Programme zu starten.

Zum Speichern von Daten (siehe die Automobildatei) nehmen binäre Files mehrere bis viele Datensätze auf, die alle den gleichen Aufbau (festgelegt mittels C - Strukturen) haben. Die Strukturdeklarationen über den Aufbau der Datensätze der Datei stehen üblicherweise in einer Headerdatei (z.B. filename.h). Alle Programme, welche die Datei nutzen wollen inkludieren diesen Headerfile, definieren passende Variable (vom Typ der Struktur) und haben damit sinnvollen Zugriff auf die Daten.

Mangels End-of-File Zeichen wird EOF aus der im Verzeichnis (Directory des Filesystems) eingetragenen Dateigröße erkannt.

Nach der Dauer ihrer Existenz unterscheidet man permanente und temporäre Files - die Lebensdauer eines temporären Files ist durch Programmterminierung begrenzt (das Programm löscht die Datei selbst).

9.3.3. Dateien aus C ansprechen

Dateien werden unter Verwendung einer Struktur vom Datentyp FILE vereinbart. FILE ist ein in stdio.h definierter Typ, der Informationen über das geöffnete File enthält:

- · Zugriffstyp(nur lesen, nur schreiben, beides)
- · Pufferungstyp(nicht, vollständig, zeilenweise)
- · Pufferadresse
- · Fehlerstatus, ...

Die 3 Standarddateien werden automatisch beim Start des Programmes zugeordnet. Normalerweise ist die Standard-Eingabedatei **stdin** die Tastatur, die Ausgabedateien **stdout** und **stderr** sind dem Bildschirm zugeordnet, durch die I/O-Umleitung auf Betriebssystemebene können auch beliebige Dateien involviert sein.

Will man vom Programm aus eine bestimmte Datei ansprechen, dann muß diese mit der Funktion fopen eröffnet werden.

FILE *fopen(const char filename[], const char modus [])

Mit Dateiname und Öffnungsmodus (lesen, schreiben) wird die Datei geöffnet bzw. die Verbindung zwischen C Programm und dem Filesystem hergestellt. Als Funktionsrückgabewert liefert fopen die Adresse (werden später noch näher erläutert) einer Struktur vom Typ FILE, die für alle weiteren Lese- und Schreiboperationen verwendet wird oder auch 0 (präziser den NULL-Zeiger), falls ein Fehler aufgetreten ist.

filename ist ein String der den externen Filenamen (eventl mit Pfad) enthält,
modus ist ein String, der die Zugriffsart angibt.

modus	Verarbeitung	Dateizeiger	EOF-Marke
r	nur Lesen (Text)	vor dem 1. Byte der Datei	unverändert
r+	lesen und schreiben (T)	"	"
rb	nur lesen (binär)	"	"
r+b od. rb+	lesen und schreiben (b)	"	"
w	nur schreiben (Text)	neu angelegt	am Anfang
w+	lesen und schreiben(T)	"	"
wb	nur schreiben (binär)	"	"
w+b od. wb+	lesen und schreiben (b)	"	"
a	nur schreiben (Text)	hinter dem letzten Byte	unverändert
a+	lesen und schreiben (T)	"	"
ab	nur schreiben (binär)	"	"
a+b od. ab+	lesen und schreiben(b)	"	"

"r" setzt eine bereits existierende Datei voraus; wenn nicht, liefert fopen() den Wert NULL
"w" überschreibt eine eventuell vorhandene Datei, da sowohl Positionsanzeiger als auch EOF-Marke auf die 1.Dateiposition gesetzt werden
"a" schreibt jede Ausgabe hinter das aktuelle Ende der Datei

```
FILE *datei;
char zeile[1000];
  ....
  if ((datei=fopen("bsp.txt","r+"))==NULL)
      perror ("Open-bsp.txt : ");     /* Fehlermeldung */
     else  ...
.....
  fgets (zeile, sizeof(zeile), datei);
```

Sobald die Filebearbeitung beendet ist, soll die Datei durch fclose() geschlossen werden. Bei Ausgabedateien wird vorher noch der Inhalt des Puffers in die Datei übertragen.

int fclose(FILE *datei); (**datei** ist der logische Dateiname) /* fclose(datei); */

Eröffnen eines temporären Files:

FILE *tmpfile(void);

Funktioniert wie fopen; der externe Filename wird automatisch so generiert, daß er nicht mit dem eines existierenden Files übereinstimmt (modus = wb+). Schließen eines temporären Files ist nicht notwendig (wird bei Programmende geschlossen).

9.3.4. I/O Befehle für Dateien

Gliederung der I/O Befehle	Eingabebefehle	Ausgabebefehle
für zeilenorientierte Dateien mit ASCII Zeichen	fgetc (fetc, getchar) fgets (gets) fscanf (scanf)	fputc (putc, putchar) fputs (puts) fprintf (printf) fvprintf (vprintf)
I/O von und nach Strings	sscanf	sprintf (svprintf)
für I/O von binären Dateien (Open-Art b)	fread ftell fgetpos	fwrite fseek fsetpos

Die nachfolgenden Beispiele verfolgen nur den Zweck, verschiedene I/O Befehle in korrekter Syntax wiederzugeben. Das erste Programm beschreibt eine Textdatei (fiobsp.1) mit 5 Zeilen. Im zweiten Programm werden diese 5 Zeilen mit äquivalenten Befehlen wieder ausgelesen.

```
#include "stdio.h"   /* 1. Beispiel zur Anwendung der C - I/O Funktionen */

/* In jeder Sprache müssen Dateien definiert und anschließend eröffnet
   werden. In C wird ein Pointer auf eine File - Struktur (enthalten in
   stdio.h) angelegt. Das geschieht mit FILE *datei1,*datei2; die Datei
   kann nun eröffnet,bearbeitet und geschlossen werden */

void main ()
{
   FILE *datei1;
   int i;

   if ((datei1 = fopen("fiobsp.1","w")) == NULL)
   {       puts ("Ausgabefile >fiobsp.1< open-Fehler");
           exit (1);
   }                   /* der File ist nun eröffnet */
   for (i=1; i<6 ; i++)
   {       fputc ('A',datei1);     /* 1. Zeichen schreiben */
           fputs ("nfang der Zeile Nummer : ",datei1);
                          /* Stringinhalt (ohne \0) schreiben */
           fprintf(datei1," %d \n",i);
                           /* Rest der Zeile schreiben */
   }
```

```
   if ((i = fclose(datei1)) != 0)
   {       printf ("File >fiobsp.1< close-Fehler %d \n",i);
           exit (1);
   }
   exit (0);
}
```

Die Ausgabe dieses Programms (geschrieben in die Datei fiobsp.1) sieht folgendermaßen aus:

```
Anfang der Zeile Nummer :  1
Anfang der Zeile Nummer :  2
Anfang der Zeile Nummer :  3
Anfang der Zeile Nummer :  4
Anfang der Zeile Nummer :  5
```

```
      /* 2. Beispiel zur Anwendung der C - I/O Funktionen */
#include "stdio.h"
#define BUFLEN 25

/* Jetzt wird die Ausgabe des vorigen Programms wieder eingelesen. Es wird
   versucht jeweils die zur Ausgabe äquivalenten Befehle zu verwenden.
   fgets liest ganze Zeilen (bis inclusive \n), aber nur eine maximale
   Anzahl von Zeichen. Hier liefert das erste fgets daher nur 24 Zeichen,
   das zweite fgets liest den Zeilenrest */

void main ()
{
   FILE *datei2;
   int  i,j,c;
   char buffer [BUFLEN];
   char *ptr_c;

   if ((datei2 = fopen("fiobsp.1","r")) == NULL)
   {       puts ("Eingabefile >fiobsp.1< open-Fehler");
           exit (1);
   }        /* der File ist nun eröffnet */
   for (i=1; i<6 ; i++)
   {       c = fgetc (datei2);          /* 1 Zeichen lesen */
           if ( c == EOF )     break;  /* zwei gleichwertige Abfragen */
           if ( feof(datei2) ) break;  /* ob EOF vorliegt */

           ptr_c = fgets (buffer,BUFLEN,datei2);
           if ( ptr_c == NULL) break;   /* NULL wird bei EOF zurückgegeben */
                       /* hier werden 24 Zeichen gelesen */
           j = fscanf(datei2,"%d",&c);   /* j enthält 1 wenn erfolgreich, */
           if ( j == EOF)      break;    /* sonst EOF                     */
                                         /* die Zahl ist eingelesen */
           fgets (buffer,BUFLEN,datei2);
           if ( feof(datei2) ) break;
                  /* jetzt wurde der Rest der Zeile (  \n) gelesen */
   }
   if ((i = fclose(datei2)) != 0)
   {       printf ("File >fiobsp.1< close-Fehler %d \n",i);
           exit (1);
   }
   exit (0);
}
```

Will man Daten wie in den oben gezeigten Datensätzen schreiben und lesen, dann hat man nicht nur charakters (Zeichen), welche in die Datei kommen, sondern auch integer und float Werte. Deshalb muß statt zeichenorientierter I/O auf binäre datensatzweise I/O gewechselt werden. Man wird die Datei mit b (binärer Übertragungsmodus) eröffnen und mit fread und fwrite jeweils ganze Datensätze lesen bzw. schreiben.

9.3.5. Lesen und Schreiben von Binärfiles

Lese und Schreibbefehl haben identisches Aussehen, es werden sinngemäß immer eine Menge von Bytes (Länge eines Datensazes * Anzahl Datensätze) geschrieben bzw. gelesen.

```
int fread (void *daten, int datensatzgroesse, int anzahl, FILE *datei);
int fwrite(void *daten, int datensatzgroesse, int anzahl, FILE *datei);
```

daten :	Adresse der ein- oder auszugebenden Variablen (&Strukturname oder Arrayname ohne [])
groesse :	Größe eines Datensatzes in Byte (sizeof())
anzahl:	Anzahl der zu lesenden bzw. zu schreibenden Datensätze
datei:	Zeiger auf die Datei, der von fopen geliefert wurde.

Funktionsrückgabewert : Anzahl der korrekt übertragenen Datensätze. Bei einem Fehler (auch bei EOF) wird die Anzahl der noch fehlerfrei übertragenen Elemente zurückgeliefert.

Neben dem Lesen und Schreiben von Datensätzen kann auch direkt auf irgendeinen Datensatz (präziser auf ein beliebiges Byte) in der Datei positioniert werden. Alle Bytes der Datei sind von 0 beginnend durchnummeriert. Will ich auf den 10ten Datensatz in der Datei positionieren (das bedeutet das nächste fread nach der Positionierung liest den 10ten Datensatz), dann positioniere ich auf das (sizeof(Datensatz)* (10-1))te Byte der Datei.

```
int fseek(FILE *datei, long  abstand, int basis);
```

Setzt die Position für nachfolgende Schreib- oder Leseaktionen. Sie wird berechnet durch basis + abstand (in BYTES !).

basis:	SEEK_SET	0	Fileanfang (vor dem 1. Zeichen)
	SEEK_CUR	1	aktuelle Positionierung in der Datei
	SEEK_END	2	Fileende (hinter dem letzten Zeichen = EOF Position)

Funktionsrückgabewert 0 für ordnungsmäßige Funktion, sonst größer 0 sonst.

```
fseek(f,0L,SEEK_SET); = fseek(f,0L,0); ≅ rewind(f) ist am Dateianfang
fseek(f,8L,SEEK_CUR);  8 Bytes vorwärts von der aktuellen Position
fseek(f,-sizeof(Buffer),SEEK_END); vom Dateiende um ... nach vor

fseek(f,0L,SEEK_END);  = Dateiende ≅ fopen (...,"a")
```

`long` **`ftell`**`(FILE *datei)` liefert die aktuelle Positionierung in der Datei (vom Anfang der Datei oder -1L im Fehlerfall (für Binärdateien).

`int` **`feof`**`(FILE *datei)` liefert TRUE, wenn der letzte Lesebefehl schon EOF erreicht hat. BSP: while (!feof(f)) ...

Das nächste Beispiel zeigt die Möglichkeiten von fread, fwrite, fseek anschaulich, verwendet aber noch keine wirklichen Datensätze (es gilt ein Datensatz = 1 Byte).

```
#include <stdio.h>  /* 3. Beispiel zur Anwendung der C - I/O Funktionen */
#define  BUFLEN 25

/* Die bisherigen Funktionen sind eher für normale Ascii Files geeignet.
   Will man aber Datensätze bearbeiten, die auch aus Feldern mit ver-
   schiedenen Datentypen bestehen, so sind die Funktionen fread/fwrite
   besser geeignet. Mit fseek besteht zusätzlich noch die Möglichkeit im
   File zu positionieren und dadurch einen Direktzugriff auf jedes Byte der
   Datei durchzuführen.  */


/*


             1         2         3
    012345678901234567890123456789012345      Spaltenzähler in Datei
    abcdefghijklmnopqrstuvwxyz0123456789      geplante Daten  */

main ()
{
   FILE *datei3,*fp;
```

```
   int  i,j,c;
   char buffer [BUFLEN];
   char *ptr_c;

   if ((datei3 = fopen("fiobsp.2","w")) == NULL) {
          puts ("Ausgabefile >fiobsp.2< open-Fehler");
          exit (1);
          }
   fputs("abcdefghijklmnopqrstuvwxyz0123456789\n",datei3);
                              /* Testdaten schreiben; \n erzeugt 0d0a */
          /* jetzt wird die Datei geschlossen und mit r+
            (input/output) wieder eroeffnet */
   if ((i = fclose(datei3)) != 0) {
          printf ("File >fiobsp.2< 1.close-Fehler %d \n",i);
          exit (1);
          }
   if ((datei3 = fopen("fiobsp.2","r+")) == NULL) {
          puts ("bei 2. (r+) open auf >fiobsp.3< open-Fehler");
          exit (1);
          }
   printf ("die Position in der Datei ist %ld \n",ftell(datei3));

   if (fseek (datei3,5L,SEEK_SET) != 0) {
          puts ("der 1. fseek (auf 5. Byte der Datei) ging schief");
          exit (1);
          }
   if ((i = fread (buffer,sizeof(char),3,datei3)) != 3) {
          puts ("der 1. fread hat nicht 3 Zeichen gelesen; boese");
          exit (1);
          }
   printf ("nach 1. fread steht   %.3s    im buffer \n",buffer);
   printf ("die Position in der Datei ist %ld \n",ftell(datei3));

   if (fseek (datei3,4L,SEEK_CUR) != 0) {
          puts ("der 2. fseek (4 Bytes ab Position) ging schief");
          exit (1);
          }
   if ((i = fread (buffer,sizeof(char),4,datei3)) != 4) {
          puts ("der 2. fread hat nicht 4 Zeichen gelesen; boese");
          exit (1);
          }
   printf ("nach 2. fread steht   %.4s    im buffer \n",buffer);
   printf ("die Position in der Datei ist %ld \n\n",ftell(datei3));

          /* vor dem fwrite korrekt positionieren  */
          /* am Ende steht im File ein \n; dieses ist am PC ein 0d0a
             im File; auf Unix Maschinen ist es nur ein 0a */

   if (fseek (datei3,-8L,SEEK_END) != 0) {
          puts ("der 2. fseek (auf 8. Byte vom Ende) ging schief");
          exit (1);
          }
   if ((i = fwrite ("-+-=",sizeof(char),4,datei3)) != 4) {
          puts ("der 1. fwrite hat nicht 4 Zeichen geschrieben; boese");
          exit (1);
          }
   printf ("die Position in der Datei ist %ld \n\n",ftell(datei3));

   if ((i = fclose(datei3)) != 0) {
          printf ("File >fiobsp.2< close-Fehler %d \n",i);
          exit (1);
          }
   exit (0);
}
```

In der nächsten Zeile steht der Inhalt der Datei fiobsp.2 nach Programmende. Danach finden Sie alle Meldungen, die dieses Programm auf stdout ausgegeben hat.

```
abcdefghijklmnopqrstuvwxyz0123-+-=89

die Position in der Datei ist 0
nach 1. fread steht   fgh    im buffer
```

```
die Position in der Datei ist 8
nach 2. fread steht   mnop    im buffer
die Position in der Datei ist 16

die Position in der Datei ist 34
```

Nach diesem inhaltlich eher sinnlosen Programm folgt jetzt ein größeres Programm, welches eine Datei mit Autodaten bearbeitet. In einer seperaten ...h Datei findet sich der Datensatzaufbau, etc. Die einzelnen Funktionen sollten als einzelne Dateien gespeichert werden (jede Datei enthält #include "auto.h"), hier folgen alle in Serie.

```
// auto.h       Headerdatei für das  Autodatei - Programm

#include <stdio.h>  // häufig benötigte includes
#include <conio.h>
#define FALSE 0     /* und defines*/
#define TRUE 1

               // Prototypen der Funktionen
void autoinsert (FILE *autodat); // neue Autos einfügen
void autolist   (FILE *autodat); // Autos einfach auflisten
void autosort   (FILE *autodat); // Datei sortieren
void autogrupp  (FILE *autodat); // einfache Liste gruppiert nach Marke/Type
void autorewrite(FILE *autodat); // Preiserhöhung aller Autos
                                 //   (lesen, ändern und rückschreiben)

struct automobil       // Aufbau der Datensatz-Struktur der Datei
{
    char    marke[21];  // Marke des Autos            VW
    char    type[21];   // Typenbezeichnung           Golf
    int     baujahr;    // Baujahr                    1991
    long    kilometer;  // gefahrende Kilometer       35000
    double  preis;      // Verkaufspreis des Autos    87000
};
```

```
#include "auto.h"     // main     enthält das Auto - Hauptmenü


void main ()
{
   int taste=0;
   FILE *autodat;

   autodat = fopen("auto.dat","r+b");  // öffne Datei
   if (autodat == 0)                   // ging nicht, rückfragen
   {  printf ("\n\nDatei auto.dat existiert nicht, neu angelegen (j,n) ?");
      taste = getch();
      if (taste == 'J' || taste == 'j')
      {  autodat = fopen("auto.dat","w");  // und eine leere Datei
         fclose (autodat);                 // wird angelegt.
         autodat = fopen("auto.dat","r+b");
      }
   }
   if (autodat == 0) exit(1); // sofortiges Ende von main (des Programm)
                              // wenn es jetzt noch keine Datei gibt


   while (taste != 'E')       // Hauptmenü anzeigen und bearbeiten
   {
      clrscr();
      gotoxy(10,3);
      printf ("Willi's AUTOHAUS   -   Hauptmenü");

      gotoxy(14,6);
      printf ("A - Autos sequentiell auflisten ");
      gotoxy(14,8);
      printf ("B - Liste mit Summenwert pro Marke/Type ");
      gotoxy(14,10);
      printf ("C - Autos sortieren nach Marke Type ");
      gotoxy(14,12);
      printf ("P - Preise der Autos um 2 %% erhöhen ");
      gotoxy(14,14);
```

```
      printf ("D - Autos erfassen ");
      gotoxy(14,18);
      printf ("E - Programm beenden");
      gotoxy(45,20);
      printf ("bitte auswählen ");
      gotoxy(65,20);

      taste = getch();
      if (taste == 0) taste = getch () + 256; /* Funktions-/Cursortaste */

      gotoxy (5,24);
      switch (taste)
      {
       case 'a' :
       case 'A' :  autolist (autodat);
                   break;
       case 'b' :
       case 'B' :  autogrupp (autodat);
                   break;
       case 'c' :
       case 'C' :  autosort (autodat);
                   break;
       case 'p' :
       case 'P' :  autorewrite (autodat);
                   break;
       case 'd' :
       case 'D' :  autoinsert (autodat);
                   break;
       case 'e' :
       case 'E' :  taste = 'E';
                   break;
       default  :  gotoxy (5,24);
                   printf ("falsche Taste - bitte Menüpunkt wählen ");
                   delay (2000); // 2 Sekunden warten, Zeit zum lesen
      }
   }
   fclose (autodat);
   clrscr();
   printf ("Bye\n\n");
}


                // neue Autos eingeben und in Datei speichern
void autoinsert (FILE *autodat)
{
    struct automobil auto1;     // sehr einfache Version der
    int taste=0;                // Datensatzeingabe

    fseek(autodat,0L,SEEK_END); // ans Dateiende stellen um
    clrscr();                   // neue Sätze hinten anzufügen
    do
    {
        printf ("Marke     eingeben :\n");
        fgets (auto1.marke,sizeof(auto1.marke),stdin);
              // entfernen des /n am Ende des eingegebenen Textes
        auto1.marke[strlen(auto1.marke)-1] = '\0';
        printf ("Type      eingeben :\n");
        fgets (auto1.type,sizeof(auto1.type),stdin);
        auto1.type[strlen(auto1.type)-1] = '\0';
        printf ("Baujahr   eingeben :\n");
        scanf ("%d",&auto1.baujahr); fflush (stdin);
        printf ("Kilometer eingeben :\n");
        scanf ("%ld",&auto1.kilometer); fflush (stdin);
        printf ("Preis     eingeben :\n");
        scanf ("%lf",&auto1.preis); fflush (stdin);

               // schreiben des Datensatzes
        fwrite (&auto1, sizeof(auto1), 1, autodat);


        printf ("\n Noch ein Auto erfassen (j,n) ? ");
        taste = getch();
    } while (taste == 'J' || taste == 'j');
}


              // einfaches Auflisten der gespeicherten Autos
```

```
void autolist (FILE *autodat)
{
    struct automobil auto1;
    int taste=0;

    fseek(autodat,0L,SEEK_SET);   // an den Dateianfang stellen
    clrscr();
    printf ("  %-20.20s  %-20.20sBaujahr     km     Preis\n\n",
                  "Marke", "Type");

    while (fread(&auto1,sizeof(auto1), 1, autodat) )
    {
        printf ("  %-20.20s  %-20.20s  %4d %7ld %12.2lf\n",
               auto1.marke, auto1.type, auto1.baujahr,
               auto1.kilometer, auto1.preis);
    }
    printf ("\n zurück zum Menü mit Tastendruck ");
    taste = getch();
}
             // Sortieren der Autos nach Marke und Type
             // es wird ein dynamisches Array angelegt,
             // alle Datensätze ins Array, sortieren, rausschreiben
void autosort (FILE *autodat)
{
    struct automobil auto1;
    struct automobil *autotab; // dient zum Anlegen einer dynamischen
                               // Tabelle (Speicherplatz zur Laufzeit)
    int anzahl;
    int i,sortiert;
    int taste=0;

    fseek(autodat,0L,SEEK_SET);   // an den Dateianfang stellen
    clrscr();
    anzahl = 0;
    while (fread(&auto1,sizeof(auto1), 1, autodat) )
          anzahl ++ ;
    if (anzahl < 2)
    {  gotoxy (5,24);
       printf ("zuwenig (%d) Datensätze zum sortieren",anzahl);
       delay (2000);   // Zeit zum Lesen
       return;         // und retour
    }

        // Speicherplatz für das Array anfordern
    autotab = calloc (anzahl,sizeof(struct automobil));
    if (autotab == 0)   // Speicherplatz holen mißlungen
    {  gotoxy (5,24);
       printf ("konnte keinen Hauptspeicher zum Sortieren ordern");
       delay (2000);   // Zeit zum Lesen
       return;         // und retour
    }
      // autotab stellt jetzt eine Tabelle mit "anzahl" Elementen dar
      // jedes Element ist vom Typ struct automobil

      // einlesen der Datei ins Array
    fseek(autodat,0L,SEEK_SET);   // an den Dateianfang stellen
    for (i=0;fread(&autotab[i],sizeof(struct automobil), 1, autodat);i++);

      // sortieren des Arrays
    sortiert = FALSE;
    while (sortiert == FALSE)
    {



        sortiert = TRUE;
        for (i=0;i<anzahl-1;i++)
            if (strcmp(autotab[i].marke,autotab[i+1].marke)>0 ||
                strcmp(autotab[i].marke,autotab[i+1].marke)== 0 &&
                 strcmp(autotab[i].type,autotab[i+1].type)>0 )
            {
                auto1 = autotab [i];        /* wenn nötig Tausch */
                autotab[i] = autotab [i+1];
                autotab[i+1] = auto1;
                sortiert = FALSE;
            }
    }
```

```
      // zurückschreiben des Arrays auf die Datei
    fseek(autodat,0L,SEEK_SET);   // an den Dateianfang stellen
    for (i=0;i<anzahl &&
              fwrite(&autotab[i],sizeof(struct automobil), 1, autodat);i++);

      // Speicherplatz wieder hergeben
    free (autotab);

    printf ("\n %d Datensätze wurden sortiert\n",anzahl);
    printf ("\n zurück zum Menü mit Tastendruck ");
    taste = getch();
}

                  // auflisten der gespeicherten Autos mit
                  // Gruppenwechsel nach Marke und Type
void autogrupp (FILE *autodat)
{
    struct automobil auto1;
    int taste=0;
    char vergleich_marke[20], vergleich_type[20]; // Vergleichsfelder
    long sumt_km = 0;                       // Summenfelder für
    double sumt_preis = 0;                  // die beiden Gruppenstufen
    long summ_km = 0;                       // t = Type
    double summ_preis = 0;                  // m = Marke

    fseek(autodat,0L,SEEK_SET);   // an den Dateianfang stellen
    clrscr();
    fread(&auto1,sizeof(auto1), 1, autodat); // lese 1. Satz
    while (! feof(autodat) )
    {          // Gruppenkopf für Marke
        summ_km = summ_preis = 0;
        printf (" Marke : %-20.20s \n", auto1.marke);
        strcpy (vergleich_marke,auto1.marke);
        while (strcmp(auto1.marke,vergleich_marke)==0 && ! feof(autodat))
        {          // Gruppenkopf für Type
            sumt_km = sumt_preis = 0;
            printf ("   Type : %-20.20s \n", auto1.type);
            strcpy (vergleich_type,auto1.type);
            while (strcmp(auto1.type,vergleich_type)==0 && ! feof(autodat))
            {          // Einzelsatzverarbeitung
                printf ("            %4d %7ld %12.2lf\n",auto1.baujahr,
                                       auto1.kilometer, auto1.preis);
                sumt_km    = sumt_km    + auto1.kilometer;
                sumt_preis = sumt_preis + auto1.preis;

                fread(&auto1,sizeof(auto1), 1, autodat); // lese Sätze
            }
                   // Gruppenfuß für Type
            summ_km    = summ_km    + sumt_km;
            summ_preis = summ_preis + sumt_preis;
            printf ("     Summe-Type: %7ld %12.2lf\n",sumt_km, sumt_preis);
}
                 // Gruppenfuß für Marke
        printf ("    Summe-Marke: %7ld %12.2lf\n",summ_km, summ_preis);
}
printf ("\n zurück zum Menü mit Tastendruck ");
    taste = getch();
}
```

```
        // lesen und rückschreiben aller Datensätze
        // alle Preise um 2 % erhöhen,
void autorewrite (FILE *autodat)
{
    struct automobil auto1;
    int taste=0;
    int anzahl;

    fseek(autodat,0L,SEEK_SET);   // an den Dateianfang stellen
    clrscr();
    anzahl = 0;
    while (fread(&auto1,sizeof(auto1), 1, autodat) )
    {
              //  in der Datei um einen Datensatz zurückpositionieren
        fseek(autodat,anzahl * sizeof(struct automobil),SEEK_SET);
        auto1.preis = auto1.preis * 1.02; // den Preis erhöhen
        fwrite(&auto1,sizeof(auto1), 1, autodat);  // rückschreiben
        anzahl ++;
              // vor dem nächsten lesen wieder neu positioniern
        fseek(autodat,anzahl * sizeof(struct automobil),SEEK_SET);
    }
    printf ("\n Es wurden %d Preise erhöht\n",anzahl);
    printf ("\n zurück zum Menü mit Tastendruck ");
    taste = getch();
}
```

Übung:

- Obwohl das obige Auto-Programm umfangreich ist kann es noch erweitert werden.

  Es sollen wahlweise 2 - 3 verschiedene Sortiervarianten angeboten werden.
  Die Eingabe soll verbessert werden (alle Daten in einer Zeile eingeben)
  Autos sollen auch wieder gelöscht werden
  Autos sollen auch geändert werden.

- Das nachfolgende Programm ist bewußt undokumentiert und unkommentiert. Sie sollen erforschen was das Programm tut, beurteilen wie sinnvoll die Lösung ist und letztendlich auch Mängel und mögliche Fehler finden. Weil sehr zahlreich von vordefinierten C - Funktionen Gebrauch gemacht wird sollten Sie die Funktionsliste im letzten Kapitel oder die Hilfe- Möglichkeit eines Compilers zu Rate ziehen.

```
#include <stdio.h>
#include <stdio.h>
#include <time.h>

void main()      /* Die Funktionsaufteilung scheint wenigsten sinnvoll */

{
    FILE *datei;
    int filelen, line_no;

    setrandomizestart();
    if ((datei = fopen("name.dat", "r")) == NULL)
    {   printf("datei fehlt.\n");
        exit(0);
    }
    filelen = find_size(datei);
    line_no = (rand() % filelen) + 1;
    pick_line(datei, line_no);
}
```

```
setrandomizestart()
{
    long timeval;
    timeval = time(0);
    srand(timeval);
}


find_size(datei)
FILE *datei;
{
    char buf[83];
    int filelen = 0;

    while (!feof(datei))
    {     fgets(buf, 82, datei);
          if (buf[0] != ' ')
             ++filelen;
    }
    rewind(datei);
    return(filelen-2);
}


pick_line(datei, line_no)
FILE *datei;
int line_no;
{
    int i=0;
    char buf[83];

    while (i<line_no)
    {    fgets(buf, 82, datei);
         if (buf[0] != ' ')
            ++i;
    }
    printf("\n%s", buf);
    fgets(buf, 82, datei);
    while (buf[0] == ' ')
    {    printf("%s", buf);
         fgets(buf, 82, datei);
    }
}
```

10. Variablen- und Funktionsdefinition (exakt)

Syntax der Variablendefinition :

[Speicherklasse] Datentyp [Typattribut] Name[=Initwert][,Name[=Initwert]]...;

10.1. Typattribute

Typattribute stellen zusätzliche Angaben dar, die bei jedem Datentyp möglich sind.

const macht die Variable zu einer Konstanten, die nur initialisiert bzw. der nur einmal ein Wert zugewiesen werden kann. Weitere Zuweisungen an diese Variable sind nicht möglich.

volatile verhindert jegliche Optimierungen in Zusammenhang mit dieser Variablen. Sinnvoll nur bei Systemprogrammierung.

Speicherklasse (static, ...) wurden schon bei den Funktionen behandelt

10.2. Datentyp

Der Datentyp bestimmt die interne Darstellung der Variablen und legt gleichzeitig den möglichen Wertebereich fest.

Obwohl bei Bedarf Konvertierungen vorgenommen werden, sollten im Interesse eines sauberen Programmierstils (bzw. möglicher Compilerdifferenzen) unterschiedliche Datentypen in Ausdrücken nicht vermischt werden bzw. nur dann vermischt werden, wenn die dabei (automatisch durchgeführten) Konvertierungen überblickt werden können. Noch wesentlich wichtiger ist die korrekte Einhaltung gleicher Datentypen bei den Funktionsrückgabewerten und bei den Parametern, weil der Compiler die Typdifferenzen nicht entdeckt, wenn er nicht durch einen Prototyp informiert wurde. Nie ohne Prototypen arbeiten und bei Funktionen mit variabler Parameterzahl (wie printf) immer den korrekten Datentyp der Parameter einhalten !!!

In der C-Norm werden die Datentypen wie folgt gegliedert:

Gliederung der Datentypen

normale Datentypen Datentyp von Variablen, die Speicherplatz belegen
- **skalare Typen** bestehen aus genau einem Element
  - **arithmetische Typen** sind für Zahlenwerte geeignet
    - **integer Typen** für ganzzahlige Werte
      - **grundlegende integer Typen** char, short, int, long und deren Varianten
      - **Bitfeld Typen** können einzelne Bitgruppen anzusprechen
      - **aufzählende Typen (enumeration)** selbstdefinierte Typen
    - **Gleitkomma Typen** float, double, long double
  - **Pointer** speichert die Adresse eines anderen Datentyps
- **nichtskalare Typen** enthalten mehrere Elemente
  - **Arrays** enthalten mehrere Elemente gleichen Typs
  - **Strukturen** enthalten Elemente [unterschiedlichen] Typs
  - **Unions** enthalten Elemente [unterschiedlichen] Typs, die redefiniert sind

unvollständige Typen unvollständige Datentypbeschreibung
- **unvollständige Arrays** ohne Dimensionsangabe (bei Parametern, Initialisierung)
- **unvollständige Strukturen** es wird keine Strukturvariable definiert
- **unvollständige Unions** es wird keine Unionvariable definiert
- **void** steht für kein Datentyp bzw. Datentyp nicht bekannt

Funktionen definiert wird der Typ des Funktionsrückgabewerts und der Parameter

In der Folge werden die einzelnen Typen noch etwas genauer behandelt, insbesondere wenn dies zu Beginn des Skriptums (Kapitel 4) noch nicht der Fall war. Die unvollständigen Typen werden nicht extra aufgeführt. Void wird bei den skalaren Typen besprochen, Pointer werden erst vor den Funktionen behandelt.

10.2.1. Grundlegende integer Typen

char stellt einen ein Byte langen integer dar, der eben auch als Code eines Zeichens betrachtet werden kann. Ein Minimum von 7 nutzbaren Bits (128 Zeichen) ist garantiert, normal werden 8 Bits genutzt. Es gibt die folgenden Varianten:
char | signed char | unsigned char

int entspricht dem Maschinenregister wie bereits am Anfang besprochen. Als Mindestgröße belegt short 2 Byte und long 4 Byte. Als sicher anzunehmen ist aber nur short <= int <= long. Als mögliche Varianten können angegeben werden:
short | short int | signed short | signed short int
unsigned short | unsigned short int
int | signed int | signed
unsigned int | unsigned
long | long int | signed long | signed long int
unsigned long | unsigned long int

10.2.2. Bitfeld Typen

Einzelne Bits werden in C üblicherweise mittels und bzw. oder Verknüpfung angesprochen. Mit der und Verknüpfung wird das Bit abgefragt, mit der oder Verknüpfung gesetzt:

```
#define BIT3  4
int schalter;                    /* statt BIT3 eher einen besseren Namen! */
   if  (schalter & BIT3)  ... ; /* True wenn das 3. Bit von rechts auf 1 */
   schalter = schalter | BIT3;  /* setzt das 3. Bit von rechts auf 1     */
   schalter = schalter & ~BIT3; /* setzt das 3. Bit von rechts auf 0     */
```

Zusätzlich gibt es noch die eher selten verwendete Möglichkeit der Bitfeld Typen, die hier an einem Beispiel gezeigt wird:

```
                          /* Es werden zwei Bytes mit einer Struktur  */
struct flags              /* aus Bits bzw. Bitgruppen belegt.         */
{  unsigned int cf   : 1;  /* cf ist das 1. Bit von rechts            */
   unsigned int      : 1,  /* das 2. Bit von rechts bleibt frei       */
               pf   : 1,  /*        bitstru.pf = 0;                  */
                    : 9,
               iopl : 2,  /*        bitstru.iopl = 3;                */
               nt   : 1,  /* jede Bitgruppe kann als verkürzter       */
                    : 1;  /* unsigned int betrachtet werden           */
}  bitstru;
```

10.2.3. aufzählende Typen

Der Typ **enum** ermöglicht es bestimmte Werte aufzuzählen und als Wertebereich einer Variablen festzulegen.

```
enum {rot,gelb,blau,schwarz} farbe;
enum wochentag {mo,di,mi,do,fr,sa,so} ;
enum wochentag var_enum;
```

Im Programm kann die Variable dann mit den definierten Werten verwendet werden. Beispielsweise:

```
if (var_enum == so) farbe = gelb;
```

Intern wird die Variable aber als integer gespeichert, die angegebenen Werte werden von 0 beginnend durchnumeriert oder können auch angegeben werden (enum sex {male=1,female=2};).
farbe = blau; printf (" farbe= %d",farbe); liefert daher 2.
Die Werte des enum Typs werden wie normale Variablennamen behandelt, es ist daher nicht mehr möglich, eine Variable mit den Namen gelb oder mi zu definieren, wenn die obigen enum Definitionen getätigt wurden. Obwohl die enum Definition geeignet wäre klarere Programme zu erstellen, ist sie selten anzutreffen. Die Folgezeile wäre durchaus sinnvoll.

```
enum bool {false=0,true=1} flag_aktion;
```

10.2.4. Gleitkomma Typen

Gleitkommatypen sind intern binär und wurden bereits am Anfang besprochen. Als mögliche Typen können angegeben werden:

```
float       /* einfache Genauigkeit */
double      /* doppelte Genauigkeit */
long double /* extra große Gleitkommazahl wenn möglich */
```

10.2.5. Void

Void heißt "kein Datentyp"

Der Typ **void** steht deshalb bei den unvollständigen Typen, weil er eigentlich gar kein Datentyp ist und daher nur bei der Funktionsdefinition (und Pointern) verwendet wird. Es wird damit vereinbart, daß die Funktion keinen Wert zurückliefert; es wird also ein Unterprogramm definiert. Bei Pointern wird void verwendet, wenn der Datentyp, auf den der Pointer zeigt unbekannt ist. Normale Variable können mit void nicht definiert werden.

10.2.6. Arrays

Arrays können mit den Typen int, char, float, enum, mit Strukturen und auch mit Pointern gebildet werden. Mehrdimensionale Arrays sind möglich.

Die Indizes beginnen immer von 0 aufwärts zu laufen.

Beachten Sie, daß es den Funktionen strcpy und strcat möglich ist, die Parameter (des Funktionsaufrufs) zu verändern, obwohl der Operator & nicht eingesetzt wurde. Spricht man ein Array an, ohne es zu indizieren, (also ohne [ ]), dann verwendet man automatisch die Adresse des Arrays bzw. einen Pointer auf das erste Element (mit Index 0). Im Abschnitt Pointer wird diese Methode noch gründlicher betrachtet.

Man kann also problemlos Arrays (aber nicht einzelne Elemente) an eine Funktion als Parameter übergeben. Es wird nur die Adresse übergeben, wodurch die Funktion (wie bei Adressübergabe üblich) in Wirklichkeit mit dem Array im Hauptprogramm arbeitet. Die Funktion erfährt aber nichts über die Anzahl oder die Ausdehnung der Dimensionen. Deshalb muß das Array (als Parameter) in der Funktion definiert werden. Lediglich die Angabe der Ausdehnung der 1. Dimension kann entfallen, weil man sowieso selbst für das Einhalten der Dimensionsgrenzen verantwortlich ist.

```
#define DIM_TABELLE 6
#define DIM1_ARRAY  3
#define DIM2_ARRAY  5
#include <stdio.h>
void main  ()
{
   int   n_tabelle=DIM_TABELLE,n1_array=DIM1_ARRAY,n2_array=DIM2_ARRAY;
   float tabelle  [DIM_TABELLE];
   int   array    [DIM1_ARRAY] [DIM2_ARRAY];
   .
   .      printf ("Die Summe aller Werte ist %d ",
                   sufunkt(n_tabelle,tabelle,n1_array,n2_array,array) );
}
float sufunkt (int n_tab,float tab [],
            int n1_mat,int n2_mat,int mat [][DIM2_ARRAY])
   /* Hier folgen wiederum unvollständige Arraydefinitionen bei den
      Parametern. Dies ist möglich, weil die Parameter ja keinen eigenen
      Speicherplatz belegen, sondern wegen Call by Adress direkt über das
      jeweilige Hauptprogrammarray redefiniert sind. */

   /*  int  mat [] [DIM2_ARRAY];     nur die erste Dimension kann */
{                                  /* weggelassen werden           */
   int i,j;
   float sum=0;   /* auto, Initialisierung bei jedem Funktionsaufruf */
   for (i=0;i < n_tab;i++) sum += tab [i];
   for (i=0;i < n1_mat;i++)
          for (j=0;j < n2_mat;j++) sum += mat [i][j];
   return (sum);
}
```

```
int strlen (char stri[])  /* Beispiel, ermittelt Länge eines Strings */
{
   int i;
   for (i = 0; stri[i] != '\0'; i++) ;
   return (i);
}
```

```
void strcpy(char s1[], char s2[])   /* kopieren String von s2 nach s1, */
                      /* der Platz der Zielvariable s1 muß im Haupt-  */
{                     /* programm ausreichend groß vorhanden sein     */
   int i;
   for (i = 0; (s1[i] = s2[i]) != '\0'; i++);
}
```

10.2.7. Strukturen

Strukturen wurden vollständig bei den Dateien behandelt.

10.2.8. Unions

Obwohl Unions genauso wie Strukturen definiert und angesprochen werden, sind sie grundverschieden. Bei Strukturen liegen die Elemente hintereinander im Speicher; bei Unions sind alle Elemente redefiniert, sie liegen also alle auf der gleichen Speicheradresse. Eine Zuweisung an ein Element verändert also (mehr oder minder planlos) auch die anderen Elemente. Unions können daher nur dann eingesetzt werden, wenn eine Redefinition benötigt wird.

```
main()
{
union {
   int value;     /* das ist der erste Teil der Union */
   struct {
      char first;   /* diese Struktur ist der zweite Teil */
      char second;  /* Beispiel für Pc, int wird mit 2 Bytes */
   } half;          /* Länge angenommen !  */
} number;

long index;

   for (index = 12;index < 300000;index += 35231) {
      number.value = index;
      printf("%8x %6x %6x\n",number.value, number.half.first,
              number.half.second);
   }
}
```

10.2.9. Pointer

Pointer sind in C Programmen sehr häufig

Wie bei einer Sprache für Systemprogrammierung zu erwarten, gibt es in C viele Einsatzmöglichkeiten bzw. Notwendigkeiten für Pointer. Wenn Sie die Call by Value Übergabe bei Funktionsaufrufen umgehen wollen, müssen Sie quasi händisch die Adressen der formalen Parameter übergeben und im Unterprogramm mit diesen Pointern arbeiten. Arraybezüge (ohne Indexangabe) stellen immer die Adresse (einen Pointer) des Arrays dar. Sehr viele Funktionen liefern oder erwarten Pointer. Zum bequemen Umgang mit Pointern gibt es auch spezielle Operatoren (&, *,->), die nur im Zusammenhang mit Pointern angewendet werden können. Mit den Pointern kann man auch rechnen, was aber etwas ungewöhnlich funktioniert. Selbstverständlich werden Pointer auch zur dynamischen Hauptspeicherplatzverwaltung benötigt.

Wichtig

POINTER in C werden nicht als bloße Adressvariable, sondern immer als Variable definiert, welche die Adressen bestimmter anderer Variablentypen aufnehmen können.

Man spricht daher immer von Pointer auf int oder Pointer auf .... . Pointer auf int bedeutet, daß auf der Adresse, die der Pointer enthält, eine int Variable steht (erwartet wird). Diese int Variable wird angesprochen, wenn man den Operator * auf den Pointer anwendet. Während man im Assembler mit der Adresse alleine auskommen kann (muß) ist es in einer Hochsprache notwendig, daß der Compiler auch den Datentyp jeder Variablen (auch einer solchen die mit Pointer adressiert wird) kennt, damit Ausdrücke korrekt bearbeitet werden können.

Einen Pointer definiert man ähnlich wie eine normale Variable, vor den Variablennamen (Pointernamen) stellt man aber einen Stern, der in diesem Zusammenhang aber kein Operator ist.

```
int   *ptr;   /* definiert einen Pointer auf int namens ptr */
float *fpt;   /* definiert einen Pointer auf float */

int a,b;

a = 12;
ptr = &a;     /* ptr enthält nun die Adresse von a      */
b   = *ptr;   /* der Wert der int Variablen, auf die der Pointer ptr zeigt
                  wird der Variablen b zugewiesen. Weil ptr als Pointer auf
                 int definiert ist, wird keine Konvertierung vorgenommen */
              /*     b = a;     hätte die gleiche Zuweisung bewirkt      */
```

```
#include <stdio.h>
void main()         /* nochmal in einem abgeschlossenem Programm  */
{
int index,*pt1,*pt2;

   index = 39;                  /* irgendein numerischer Wert */
   pt1 = &index;                /* die Adresse der Variablen index */
   pt2 = pt1;
   printf("Der Wert ist %d %d %d\n",index,*pt1,*pt2);
   *pt1 = 13;           /* der Wert von index wird verändert */
   printf("Der Wert ist %d %d %d\n",index,*pt1,*pt2);
}
```

Spezielle Operatoren für die Pointer (& * ->)

Pointern kann man mit dem & Operator Adressen von anderen Variablen zuweisen. Auch viele Funktionen liefern einen Pointer als Funktionswert zurück. In Sonderfällen (bei extrem systemnaher Programmierung) kann man auch absolute Adressen zuweisen. Der Pointer gilt als leer (zeigt nirgendwo hin) wenn er NULL enthält. Wie man in "stdlib.h" nachlesen kann, gilt #define NULL ((void *) 0). Die "0" ist hier als absolute Adresse zu verstehen, "(void *)" ist ein cast Operator, der die integer 0 in einen Pointer konvertiert, der auf void zeigt. Pointer auf void stellen einen Sonderfall dar, sie zeigen auf keinen bzw. auf einen noch unbekannten Datentyp. Der Operator * kann nicht auf einen void-Pointer angewendet werden. Der void Pointer hat damit die Funktion eines Zwischenspeichers von Adressen und wird hauptsächlich dann angewendet, wenn (noch) nicht bekannt ist, auf welchen Datentyp der Pointer zeigt. Ein void-Pointer kann mittels cast Operator in einen anderen Pointer umgewandelt werden (char-Pointer = (char *) void-Pointer).

Ein Pointer ist auf Unix Maschinen und PCs mit Small-Modell identisch mit einem integer. Deshalb gab es in der Vergangenheit gelegentlich unseriöse Programme, es wurden Pointer in integer Variablen abgespeichert. Pointer, die auf unterschiedliche Datentypen zeigten, wurden vermischt, etc. Pointer sind aber generell eigene Variable (Auf PCs, im Large-Modell sind Pointer 4 Byte, ein int aber nur 2 Byte lang). Jeder Pointer muß exakt definiert sein (auf den richtigen Datentypzeigen); der Pointer darf nicht als Pointer auf char definiert sein, wenn im Speicher tatsächlich ein integer steht.

Wenn Sie mit der Funktion scanf Variable einlesen wollen, so dürfen Sie diese Variable nicht normal als Parameter übergeben. Vielmehr muß mit dem Operator & die Adresse der Variablen übergeben werden. So ist garantiert, daß scanf in den Parametern auch Werte zurückliefern kann. Das gilt natürlich für jeden anderen Funktionsaufruf, wo Sie über die Parameter Werte zurück haben wollen.

```
int i,j;
scanf ("%d",&i); /* Die Funktion scanf erwartet Adressen der Variablen */
scanf ("%d",&j); /* %d bedeutet integer lesen und dorthin stellen, wo
                        der pointer auf int (&j) gerade hinzeigt  */
```

Diese Methode eine Call by Adress zu erzwingen, kann man sich bei Arrays ersparen, weil diese ohnehin als Adresse des ersten Elements gelten (wenn kein Index angegeben wird). Werden bei mehrdimensionalen Arrays hinten Indizes weggelassen, dann gilt dies ebenfalls als Adresse im Array. Jeder Arrayname, der nicht über alle Indexangaben (eine pro Dimension) verfügt, gilt als Pointer auf den Datentyp der Arrayelemente.

Im Zusammenhang mit den Arrays muß jetzt auch die Pointerarithmetik besprochen werden, weil diese die Grundlage der speziellen Arrayindizierung ist.

Neben den typischen Pointeroperatoren (* -> &) können auch arithmetische Operatoren (++ -- + -) und der Vergleichsoperator (==) auf Pointern angewendet werden. Addiert man aber 2 zu einem Pointer auf int, so wird dieser nicht um 2 Byte erhöht, sondern um 2 mal die Länge des Datentyps, auf den der Pointer definiert ist.

es gibt eine spezielle Pointerarithmetik

```
int *pointer;
pointer++;          /* erhöht den Pointer um   sizeof(int)  Bytes  */
```

Wichtig

Da es normal nicht übermäßig sinnvoll ist, einen Pointer quer durch den Hauptspeicher zu addieren, wird diese Methode bei Arrays angewendet. Wird ein Pointer, der auf das erste Element zeigt, um 1 (also um die Anzahl Bytes der Elementlänge) erhöht, so zeigt er auf das zweite Element.

```
int i, ar[7], *p_a;

p_a = ar;                /* ar ist Adresse des ersten Elements (&ar[0]) */
for (i = 0;i < 7;i++)
{  *p_a = 0;             /* auch so werden alle Elemente mit 0 gefüllt  */
   p_a++;
}
for (i = 0;i < 7; i++)
      *(p_a + i) = 0;      /*  alternative for-Schleife  */
```

```
#include <stdio.h>
void main()
{
   char strg[40],*there,one,two;
   int *pt,list[100],index;

   strcpy(strg,"Das ist ein character String.");

   one = strg[0];                /* zwei gleichwertige Zuweisungen */
   two = *strg;
   printf("Die erste Ausgabe ist %c %c\n",one,two);

   one = strg[8];               /* noch zwei gleichwertige Zuweisungen */
   two = *(strg+8);
   printf("Die zweite Ausgabe ist %c %c\n",one,two);

   there = strg+10;        /* strg+10 ist gleichwertig zu strg[10] */
   printf("Es folgt Ausgabe 3 : %c\n",strg[10]);
   printf("Die 4. Ausgabe ist %c\n",*there);

   for (index = 0;index < 100;index++)
      list[index] = index + 100;
   pt = list + 27;
   printf("Ausgabe 5 : %d\n",list[27]);
   printf("Ausgabe 6 : %d\n",*pt);
}
```

So ähnlich könnten die String-funktionen aussehen

```
size_t strlen(char *p_a)             /* Variante 1 */
{
   size_t i;        /* size_t entspricht einem unsigned int */

   for (i=0; *p_a != '\0'; p_a++) i++;
   return (i);
}
```

```
size_t strlen(char *p_a)       /* Variante 2 */
{
   size_t i = 0;

   while (*p_a++) i++;
   return (i);
}
```

```
char *strcpy (char *s1,cgar *s2)           /* Variante 1 mit Pointern */
{
   char *ptr_ret = s1;
```

```
   while( (*s1 = *s2) != '\0' ) {
          s1++;  s2++;
   }
   return (ptr_ret);
}
```

```
char *strcpy (char *s1,char *s2)              /* Variante 2 mit Pointern */
{
   char *ptr_ret = s1;

   while (*s1++ = *s2++);
   return (ptr_ret);
}
```

Wenn also das folgende Array definiert ist: **int array[5];** dann gilt :

für das	die Adresse ermitteln	den Inhalt ansprechen
1. Element	array	*array
	&array[0]	array[0]
2. Element	array + 1	*(array + 1)
	&array[1]	array[1]
3. Element	array + 2	*(array + 2)
	&array[2]	array[2]
4. Element	array + 3	*(array + 3)
	&array[3]	array[3]

Bei Strukturen werden Pointer normal definiert, statt (*struname).element kann aber struname->element verwendet werden.

Pointer auf Strukturen

```
#include <stdio.h>
void main()
{
struct person {
   char initial;
   int age;
   int grade;
   };
struct person  kids[12],*point;

int index;

   for (index = 0;index < 12;index++) {
      point = kids + index;    /* Pointerarithmetik ! */
      point->initial = 'A' + index;
      point->age = 16;
      point->grade = 84;
   }
   kids[3].age = kids[5].age = 17;
   kids[2].grade = kids[6].grade = 92;
   kids[4].grade = 57;




   for (index = 0;index < 12;index++) {
      point = kids + index;
      printf("%c is %d years old and got a grade of %d\n",
             (*point).initial, kids[index].age,
             point->grade);
```

```
    }
}
```

Auswerten der Kommandozeile beim Aufruf

Eine spezielle Pointeranwendung macht es möglich, aus der Kommandozeile (beim Programmaufruf) Argumente zu übernehmen. Ruft man also ein fertig compiliertes und gelinktes Programm (z.B. a.out, ---.exe) auf, dann können neben dem Programmnamen Werte mitgegeben werden, die vom Betriebssystem aufgenommen und an das aufgerufene Programm weitergegeben werden.

Will man diese Werte im Programm nutzen, so muß die Funktion main folgendermaßen aussehen.

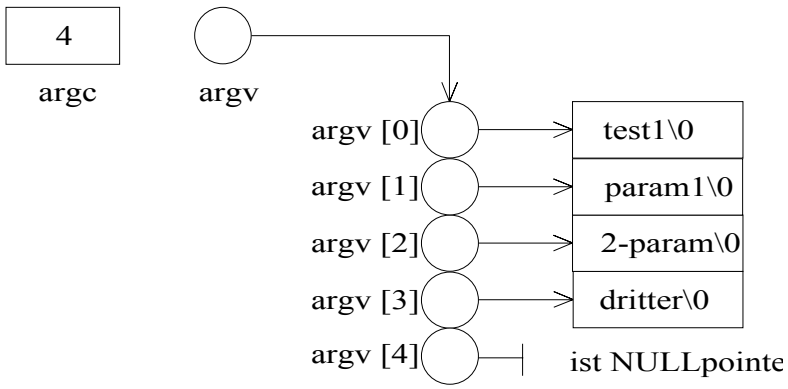
```
#include <stdio.h>
int main (int argc, char *argv[])/* Programm gibt Argumente mehrzeilig aus*/
                                  /* argc ist die Anzahl der Argumente */
                                 /* argv ist ein Array von Pointern (mit argc
                                    Elementen) jeder Pointer zeigt auf einen
                                    String, der ein Argument enthält */
{
   int i;

   for (i=1;i < argc; i++)
          printf ("%s\n",argv[i]);
   return 0;    /* Rückgabewert ans Betriebssystem (Errorlevel) */
}
```

Die Variable argv ist kein mehrdimensionales Array, sondern ein Vektor von Pointern, die jeweils die Adressen der Strings enthalten, in denen die Argumente stehen. Als erstes Argument (Index 0) gilt der Programmname.

Ruft man nun ein Programm namens test1 folgendermaßen auf:

test1 param1 2-param dritter

so sind nach dem Aufruf die Variablen mit folgenden Werten gefüllt:



argv ist (wie bei allen Arrays) eine Adresskonstante, die auf den Anfang des Arrays zeigt.

argv [0] ist das erste Element der Pointertabelle und zeigt dorthin, wo der String test1 steht.

Weil argv hier einen Pointervektor darstellt, wird mit argv[0] nur ein Pointer auf den String test1 angesprochen. Würde man nur argv verwenden, dann erhält man nur einen Pointer auf den Beginn der Pointertabelle. Will man hingegen direkt das t von test1 ansprechen, dann muß *argv[0] verwendet werden. Es gilt also:

Ausdruck	Datentyp	Inhalt ist
argv	char **	Pointer auf erstes Element des Pointerarrays
argv [0]	char *	Pointer auf den String, der den Programmnamen enthält
*argv[0]	char	erstes Zeichen des Programmnamens
**argv	char	ebenfalls erstes Zeichen des Programmnamens
argv[0][0]	char	auch erstes Zeichen des Programmnamens
*argv[3]	char	ist das d des letzten Arguments ("dritter")
**(argv+3)	char	liefert ebenfalls das d des letzten Arguments
((argv+3)+2)	char	liefert nun das i aus dem letzten Argument

In diesen Beispielen werden * und [] beliebig ausgetauscht. Das ist beim Zugriff auf Tabellen bzw. Pointerkonstrukte zulässig weil der Compiler sowieso alle [] auf * Zugriffe umsetzt. Bei der Definition von Arrays bzw. Pointer auf Arrays dürfen aber * und [] keinesgwegs ausgetauscht werden. Allzuleicht definiert man statt wirklichen Speicherplatz bloß einen Pointer. Beachten Sie daher:

```
int  tab [3] [2];       definiert ein 2-dimensionals Array mit Speicherplatz für 6 integer
int *tab[3];            definiert eine Tabelle mit Speicherplatz für 3 Pointer auf integer
int **tab;              definiert nur einen einzigen Pointer auf Pointer auf integer !!
char *texte[] =       { "eine beliebige Textzeile",
                        "dies ist ein Array mit verschiedenen Textzeilen",
                        "allerdings sind die Texte String-Konstante, das Array enthält",
                        " (durch Initialisierung) nur die Adressen der Text-Konstanten",
                        "die Dimensionsausdehnung ergibt sich durch die Initialisierung"
                      };  /* Diese Definition ist selbsterklärend */
```

dynamisches Anfordern von Hauptspeicher zur Programmlaufzeit

Wichtig

Pointer werden in C auch oft in Zusammenhang mit dynamischer Speicherplatzverwaltung genutzt. Dabei wird zur Laufzeit eine bestimmte Menge Bytes an Hauptspeicher angefordert (Funktion malloc), auf die man nur mittels Pointer zugreifen kann. Dies ist ein typischer Programmieransatz, wie er zur Entwicklung komplexer Systeme notwendig ist, aber von Sprachen wie Cobol oder Fortran nicht geboten wird.

```
/* Dieses Programm liest eine Datei ein und gibt sie, versehen mit einer
   Zeilennummerierung, wieder aus. Die I/O erfolgt über stdin/stdout,
   als Zeilenlänge sind 80 Zeichen vorgesehen.
   Damit das Programm trotzdem nicht zu trivial wird, soll die Zeilen-
   nummerierung nur in der minimal notwendigen Breite, aber trotzdem in
   sauberer Spaltenausrichtung durchgeführt werden. Also vorher die
   höchste Nummer ermitteln, dann erst mit der Ausgabe starten. Ein
   zweimaliges Lesen der Eingabe wäre nur bei Dateien möglich.         */
#include <stdlib.h>       /* in jedem Programm (hier nötig für malloc) */
#include <stdio.h>        /* immer bei I/O (fgets, printf)             */
#include <string.h>       /* bei str... Funktionen (z.B. strncpy)      */
#include <math.h>         /* mathemat. Funktionen (z.B. sqrt pow)      */
#define ZEILENLAENGE 81   /* so kann die Zeilenlänge später leicht     */
                          /*  geändert werden                          */
void main ()
{
    struct stru_elem {
       struct stru_elem   *ptr_next; /* Pointer auf Struktur stru_elem */
       char               text [ZEILENLAENGE];   /* Platz für Zeile   */
    };
    unsigned long         satzzahl = 0;
    int                   i;
    struct stru_elem      *ptr_anker = NULL ,*ptr_work,*ptr_aktuell;
    char                  buffer[ZEILENLAENGE];

while (fgets(buffer,sizeof(buffer),stdin) != NULL)
{  satzzahl++;
   /* dynamisches Bereitstellen von Hauptspeicher für ein stru_elem */
   ptr_work = (struct stru_elem *)
                  malloc (sizeof (struct stru_elem) );
   if (ptr_work == NULL)    /* Fehler beim malloc */
   {    printf (" Speicherplatz ist nicht (mehr) verfügbar !\n");
        exit (1);  /* Ende und exit_code ans Betriebssystem */
   }
   strncpy (ptr_work->text,buffer,(size_t) ZEILENLAENGE);
   if (ptr_anker == NULL)    /* Aktion beim 1. Satz */
   {    ptr_anker  = ptr_work;
        ptr_aktuell = ptr_work;
   }
   else                      /* bei den Folgesätzen */
   {    ptr_aktuell->ptr_next = ptr_work;
        ptr_aktuell = ptr_work;
   }
}
ptr_aktuell->ptr_next = NULL; /* Abschluß der Pointerkette */

for (i=1;i<20;i++)            /* wieviel-stellig ist satzzahl ? */
     if (satzzahl / (int) pow (10,i) == 0)      break;

for (ptr_aktuell = ptr_anker,satzzahl = 1;
      ptr_aktuell != NULL;
      ptr_aktuell = ptr_aktuell->ptr_next,satzzahl++)
     printf ("%*lu %s\n",i,satzzahl,ptr_aktuell->text);
                  /* i gibt die minimale Breite der Formatierung der
                     Zeilennummer an !   %*long unsigned Formatierung */
exit (0);
}
```

Hier folgt ein ähnliches Beispiel, es wird aber eine doppelt verkettete Liste erzeugt, auch mit dem Hauptspeicher wird sparsamer umgegangen.

```
/* Dieses Programm liest eine Datei (Dateiname als Aufrufparameter) ein,
   speichert diese zeilenweise in einer vorwärts und rückwärts verketteten
   Liste (Speicherplatz für die Listenelemente wird dynamisch angefordert)
   und gibt die Zeilen dann in verkehrter Reihenfolge aus, wobei die Rück-
   wärts-Pointerkette verwendet wird.
   Zeilenende wird durch \n oder EOF erkannt,
   Zeilen über 500 Bytes werden geteilt.
   Der Name der Eingabedatei kann als Parameter beim Aufruf angegeben
   werden; wenn dieser fehlt,dann wird  stdin  verwendet.

   Aufrufsyntax:        progname [dateiname]                              */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main (int argc,char *argv[])
{
   FILE *fpin;   /* Deklaration des Eingabefiles (File-Pointer) */
   char buffer [502];
   int  c=0, i=0;
   size_t bytezahl;  /* eine unsigned int Variable */

          /* struct dyn stellt den Aufbau eines Elements der Liste dar */
   struct dyn {    struct dyn *next;      /* Vorwärtspointer  next ---> */
                      struct dyn *prior;  /* Rückwärtspointer prior<--- */
                      char zeile[502]; }; /* Elementinhalt              */

   struct dyn *anker_next,*anker_prior,*w_ptr,*w_ptr_vorher;

   anker_next = anker_prior = w_ptr = w_ptr_vorher = NULL;

   if (argc > 2 ) {
          printf("ungültige Anzahl von Argumenten\n");
          printf("Aufruf mit  progname [dateiname] \n");
          exit (1);
   }
   if (argc == 2) {    /* Input - Filename im 1. Parameter */
          if ((fpin = fopen(argv[1],"r")) == NULL) {
                 printf("Datei %s nicht vorhanden ?? \n",argv[1]);
                 exit (1);
          }
   }
   else fpin = stdin;   /* stdin ist bereits eröffnet */
   while (c != EOF) {
          c = fgetc(fpin);

          if ( c == EOF && i == 0 && anker_next == NULL) {
                 printf ("leere Eingabe, daher keine Ausgabe\n");
                 exit (1);
          }
          if (c == EOF || c == '\n' || i > 500) {
                 /* ein Fall fuer Zeilenende */
                 if (c == EOF && i == 0) break;
                    /* EOF liegt direkt am Zeilenende, also fertig */
                 buffer [i++] = '\n';
                 buffer [i]   = '\0';
                    /* in buffer steht jetzt eine ganze Zeile    */
                    /* i enthält die Anzahl Zeichen in der Zeile */

                    /* jetzt wird Speicherplatz angefordert */
                    /*     (aber nur soviel wie nötig !)    */
                 bytezahl = 2 * sizeof(struct dyn *) + i + 1;
                 w_ptr = (struct dyn *) malloc (bytezahl);
                 if (w_ptr == NULL) {
                        printf ("Fehler bei malloc, kein Speicher");
                        exit(1);
                 }

                 w_ptr->next  = NULL;        /* für Ende   next---> */
                 w_ptr->prior = w_ptr_vorher;/* weiterer  prior<--- */
                 strcpy(w_ptr->zeile,buffer);
```

```
                    if (anker_next == NULL)     /* beim 1.Mal next---> */
                          anker_next = w_ptr;
                    if (w_ptr_vorher != NULL)   /* weiterer   next---> */
                          w_ptr_vorher->next = w_ptr;

                    w_ptr_vorher = w_ptr;

                    if (i > 500) buffer[0] = c, i = 1;
                    else i = 0;
             }
             else buffer[i++] = c;
      }

      anker_prior = w_ptr_vorher;    /* der Ankerpointer für   prior<--- */

      /* also los mit der Ausgabe
         Die Abarbeitung einer Pointerkette ist relativ einfach; man
         startet mit dem Anker-Pointer und setzt solange dieser nicht
         NULL ist (nirgendwohin zeigt) mit dem jeweils nächstem Pointer
         in der Kette fort. Jedes Element der Liste verfügt deshalb über
         einen Pointer pro Pointerkette                                  */

      for (w_ptr = anker_prior;
           w_ptr != NULL;
           w_ptr = w_ptr->prior)  printf ("%s",w_ptr->zeile);

      exit (0);  /* exit beendet Prog.,liefert exit-Code an Betriebssystem */
}
```

10.2.10. Funktionen

Modularisierung durch Funktionen

In keiner anderen Programmiersprache wird den Funktionen und damit der Modularisierung des Gesamtprogramms in kleine überschaubare Einheiten soviel Bedeutung beigemessen wie in C.

Wichtig

Modularisierung gehört zur Philosophie der Sprache und wird auch von den meisten C - Programmierern eingehalten. Seien Sie keine Ausnahme. Die Compiler selbst verfügen auch über eine effiziente Parameterübergabe (den sogenannten Stack), damit Funktionsaufrufe zeitlich optimiert ablaufen.

Die Modularisierung ist dann sinnvoll, wenn die Schnittstellen zwischen den Modulen so **klein** wie möglich gehalten werden. Nur dann können die Module als logisch eigenständige Einheiten betrachtet und auch als solche getestet werden. Die Dokumentation muß zuallererst eine exakte Beschreibung dieser Schnittstellen (Parameter, extern-Variable) enthalten. Falsche Inhalte in den Schnittstellendaten dürfen die Funktion selbstverständlich nur zu einer korrekten Fehlerrückmeldung veranlassen. Es darf nie vorkommen, daß ein Unterprogramm (Funktion) mit nicht korrekten Rückgabeparametern ins rufende Programm zurückkehrt.

Es ist notwendig, in der rufenden Funktion (dem Hauptprogramm) die Prototypen aller aufgerufenen Funktionen (Funktion bzw. Unterprogramm) zu vereinbaren.
Damit wird in der rufenden Funktion vereinbart, welcher Datentyp von der aufgerufenen Funktion zurückgegeben wird. Ebenso wird in den neueren Ansi-C Compilern Typ und Anzahl der Parameter vereinbart. Es ist aber nicht möglich, Anzahl oder Typ der Parameter im Prototyp zu deklarieren, wenn die Funktion eine variable Anzahl Parameter hat (printf). Wenn ein Prototyp existiert, dann kann der Compiler erkennen, ob die Datentypen von Funktionsrückgabewert und Parametern übereinstimmen und wenn nötig eine automatische Typconversion durchführen. Wenn aber kein Prototyp existiert bzw. für die Parameter kein Prototyp möglich ist, dann wird als Funktionsrückgabewert int angenommen, die Parameter werden übergeben wie sie eben vorliegen. Es liegt dann in **Ihrer** Verantwortung, daß jede Funktion nur mit Parametern im korrekten Datentyp aufgerufen wird. Es kann bereits von entscheidender Bedeutung sein, ob man 0 oder 0L verwendet.

Funktionen können die Werte in den üblichen skalaren Datentypen, aber auch Pointer aller Art zurückgeben. Wenn eine Funktion per Definition nichts zurückgeben soll, dann wird void angegeben; damit ist die Funktion ein Unterprogramm. Schreibt man keinen Datentyp vor die Funktion, dann wird per Default int angenommen.

Funktionen haben normalerweise das Speicherplatzattribut extern, durch Angabe von static ist auch ein extern-static (nur innerhalb des Files gültig) möglich.

```
   void upro ();  /* Prototyp im alten C, kein Datentyp für Parameter */
   main ()
   {    /* kurzes Hauptprogramm für upro - Aufruf */
          int x; char y;
          ..................... upro (&x,y);
   }
   void static upro (a,b)   /* Parameternotation im alten C */
   int    *a;
   char   b;
   {              /* dies ist eine Funktion ohne Rückgabewert (void!)    */
                  /* die Gültigkeit erstreckt sich nur auf den gemeinsam
                     compilierten Code (static!)                         */
                  /* beendet wird die Funktion in diesem Fall durch das
                     Ende des Funktionsblock (}), es wäre auch die return-
                     Anweisung ohne Rückgabewert denkbar.                */
                  *a = 0;
                  if ( isprint(b)) *a = b;
   }
```

Die Definition von Funktionen ist immer klar, weil die Schachtelung von Funktionen nicht erlaubt ist. Nach dem Ende einer Funktion können nur externe Variablen oder eben eine weitere Funktion kommen.

```
void      upro2 (int);/* korrekte Prototypen, mit Funktionsrückgabewert */
char      fun1();     /*                      und Typ der Parameter     */
int       *fun2(int);
int       printf(const char*, ...); /* 1. Parameter fix, weitere variabel */
int main (int argc,char *argv[]) /* Funktion main, korrekt nach C-Norm  */
{                              /* der Rückgabewert entspricht exit(.) */
...
}

void upro2 (a)    /*  Funktion upro1, ein Parameter, kein Rückgabewert   */
int a;            /*  Parameterschreibweise altes C                      */
{
...
}

char fun1 ()    /*  Funktion fun1, kein Parameter, Rückgabewert char */
{
...
   return 'x';
}

int *fun2 (int x)/*  Funktion fun2, ein Parameter, Rückgabewert ist ein */
{                                          /*   Pointer auf integer  */
int  zahl;
...
   return &zahl; /*  syntaktisch ist diese Rückgabe wohl korrekt, aber ? */
}
```

Neben der Deklaration von Funktionen gibt es auch Pointer auf Funktionen.

char	fun1 ();	Deklaration der Funktion fun1, Rückgabewert char
int	*fun2 ();	Deklaration von fun2, Rückgabewert Pointer auf integer
void	upro2 ();	Deklaration der Funktion upro2, kein Rückgabewert
int	*funp;	weil die () fehlen ist funp ein normaler Pointer auf int
int	(*funptr) ()	das ist die Definition eines Pointers (Name funptr) der auf Funktionen zeigt, welche integer als Rückgabewert haben.
int	*(*fffpt) ()	am schönsten ist wohl dieser Pointer auf Funktionen, welche als Rückgabewert einen Pointer auf integer liefern

Pointer auf Funktionen ermöglichen sehr flexible Programmierung

Einen Pointer auf eine Funktion kann man nutzen, wenn man beispielsweise schon im Hauptprogramm steuern möchte, welche dritte Funktion in einem aufzurufendem Upro verwendet werden soll. Im folgenden Kurzbeispiel wird von der Procedur "haupt" ein Pointer auf eine Funktion an die Funktion "upro" übergeben.

Wichtig

```
void upro (int (*fp) (FILE *));
haupt ()
{
   int (*funptr) (FILE *);


   funptr = fgetc;
   upro (funptr);
}
```

```
upro (int (*p1) (FILE *))
{       int c;
        FILE * fp;
        /* Aufruf von fgetc */
        c = (*p1) (fp);
        ...
}
```

Ein eher praxisnahes Beispiel ist die Verwendung der Funktion qsort aus stdlib.h. Diese Funktion sortiert bestimmte Elmente in einem Array, welches als Parameter übergeben wird. Ebenfalls als Parameter übergeben wird ein Pointer auf eine Vergleichsfunktion, die man selbst schreiben muß.

```
void qsort (void *base,          /* Startadresse des Array     Prototyp*/
            size_t nelem,        /* Anzahl Elemente        */
            size_t size,         /* Länge eines Elements   */
            int (*cmp) (const void *e1, const void *e2)); /* Vergleichsf.*/

int numcmp (void *v1, void *v2) /* dies ist die Vergleichsfunktion   */
{                               /* für einen integer Vergleich       */
int *x1, *x2;
   x1 = (int *)v1;  x2 = (int *)v2; /* void pointer werden zu int Pointer */
   if (*x1 < *x2)
       return -1;
   else if (*x1 > *x2)
       return 1;
   else
       return 0;
}
...
int tab [232];
...
qsort ( tab, 232, sizeof (int), numcmp );  /* sortiert die Tabelle tab */
...
/* in qsort wird folgendes stehen: if ((*cmp) (ptr1,ptr2) < 0) ... */
```

10.3. Konvertierungen zwischen Datentypen

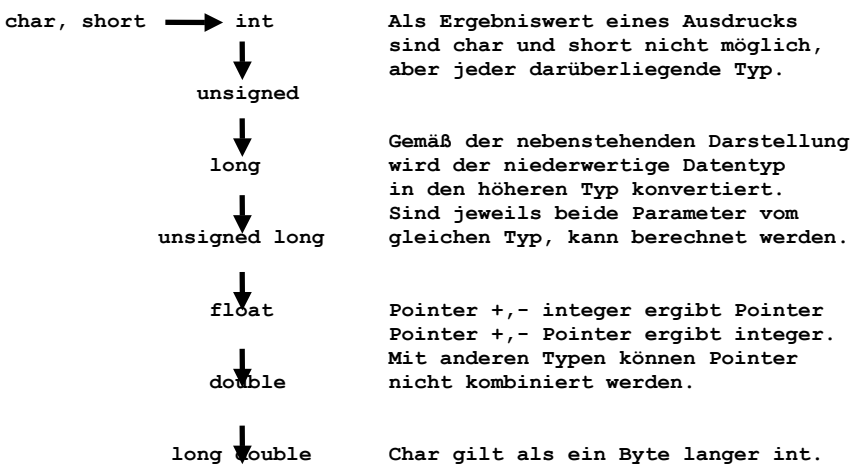
Konvertierungen bei

In C werden viele Konvertierungen auch automatisch durchgeführt. Reicht das nicht, oder will man automatische Konvertierungen generell vermeiden, so ist mit Hilfe des cast Operators jede programmgesteuerte Konvertierung möglich.

Berechnung von Ausdrücken

Während der Berechnung von Ausdrücken (bei der Abarbeitung der binären Operatoren +,-,*, ...) werden folgende Konvertierungen vorgenommen:

```
   im allgemeinen ist das Ergebnis vom Typ des höherwertigen Operanden,
   jedoch mindestens vom Typ int (keine Berechnung in short, char).
```


char, short
int
unsigned
long
unsigned long
float
double
long double
Als Ergebniswert eines Ausdrucks sind char und short nicht möglich, aber jeder darüberliegende Typ.
Gemäß der nebenstehenden Darstellung wird der niederwertige Datentyp in den höheren Typ konvertiert. Sind jeweils beide Parameter vom gleichen Typ, kann berechnet werden.
Pointer +,- integer ergibt Pointer
Pointer +,- Pointer ergibt integer.
Mit anderen Typen können Pointer nicht kombiniert werden.
Char gilt als ein Byte langer int.

Zuweisungen

Bei Zuweisungen wird jeweils der vorhandene Typ in den Zieltyp konvertiert. Dabei ist folgendes zu beachten:

Das niederwertigste (rechte) Byte eines short oder long Typs wird direkt zum char. Umgekehrt werden die linken Bits des int - Typs mit 0 aufgefüllt. Man erhält im integer den jeweiligen Code des char- Zeichens.

Unterschiedliche short, long, int Längen werden duch Unterdrückung oder Auffüllung der linken Bits konvertiert. Unsigned, signed wird entsprechend angepaßt.

Double wird durch Rundung nach float konvertiert, ebenso long double nach double.

Float wird aber durch abschneiden nach int bzw. long konvertiert.

Bei Pointern gilt generell: Nur Pointer auf den jeweils gleichen Typ und Pointer auf void sollten einander zugewiesen werden. Ein integer kann in einen Pointer umgewandelt werden und umgekehrt.

Funktionsaufrufe

Wenn beim Funktionsaufruf Parameter übergeben werden, so erfolgt immer Call by Value. Die Variablen des rufenden Programms werden (vor dem wirklichen Aufruf) in den sogenannten Stack übertragen. Dort werden sie dann von der aufgerufenen Funktion verwendet. Ist ein Prototyp angegeben, dann sind die Datentypen am Stack identisch mit denen des Prototyps. Sollten Unterschiede zu den Datentypen beim Funktionsaufruf vorliegen, dann erfolgt die Konvertierung wie bei der Zuweisung. Gibt es keinen Prototyp, so ist der Typ der Variablen am Stack identisch den Parametertypen im Hauptprogramm, es werden aber die folgenden zwei Konvertierungen getätigt, die auch automatisch bei der aufgerufenen Funktion berücksichtigt werden (nur wenn diese in alter C - Notation geschrieben wurde).

char, short	werden zu	int
float	wird zu	double

11. Anhang

11.1. Die C Bibliotheken

Was immer in C als Elemente der Sprache nicht realisiert ist, gibt es in Form von Funktionsbibliotheken (wie beispielsweise die I/O). Jede Bibliothek besteht aus einer Library mit den Funktionen als Object-Code und einer dazugehörigen Include Datei (xxx.h), in der die Funktionsprototypen, Strukturdefinitionen und ähnliches mehr stehen. In der C-Norm sind die folgenden Bibliotheken vorgesehen:

assert.h	stellt das Macro assert(Bedingung) als leicht abschaltbare Testhilfe zur Verfügung.
ctype.h	Code Type; stellt die Gruppe der Funktionen is--- (isdigit, isupper, toupper,...) bereit.
errno.h	ermöglicht den Zugriff auf die interne Fehlernummer und definiert einige Fehlertypen
float.h	definiert diverse Werte (Wertebereich, Stellenanzahl) von Gleitkommafeldern
limits.h	definiert Wertebereiche für die grundlegenden integer Typen (auch char)
locale.h	länderspezifische Angaben (z.B. isupper erkennt auch Ä, Ö, Ü)
math.h	stellt diverse mathematische Funktionen bereit
setjmp.h	nur für interne Puffersetzung relevant
signal.h	defines und Funktionen für die Signalbehandlung. Am PC praktisch nur für CTRL-C
stdarg.h	notwendig, wenn man selbst Funktionen mit variabler Parameterzahl schreibt
stddef.h	enthält allgemeingültige Definitionen wie size_t, wchar_t, NULL
stdio.h	stellt defines und Funktionen für die Ein- Ausgabe bereit
stdlib.h	enthält Prototypen für grundlegende Funktionen (atoi, malloc, getenv, exit, system, ...)
string.h	Funktionen zur Stringbehandlung (strncpy, strrchr, strtok, memmove, ...)
time.h	Funktionen und Strukturen zur Zeitbehandlung (struct tm, time, clock, ...)

In der Folge werden einige wichtige Funktionen näher erläutert.

11.1.1. ctype.h

Diese Funktionsgruppe prüft Zeichen ob sie zu einer bestimmten Gruppe gehören. Weil die länderspezifischen Angaben noch nicht bei allen Compilern korrekt sind, kann es zu Problemen bei deutschen Umlauten kommen.

int isalnum (int c)	prüft c auf alphabetisch oder numerisch. Rückgabewert true oder false.
int isalpha (int c)	prüft c auf alphabetisch. Rückgabewert true oder false.
int iscntrl (int c)	prüft c ob es ein CTRL-x Zeichen ist. Rückgabewert true oder false.
int isdigit (int c)	prüft c auf numerisch. Rückgabewert true oder false.
int isgraph (int c)	prüft c ob es druckbar aber kein blank ist. Rückgabewert true oder false.
int islower (int c)	prüft c ob es ein Kleinbuchstabe ist. Rückgabewert true oder false.
int isprint (int c)	prüft c ob es ein druckbares Zeichen ist. Rückgabewert true oder false.
int ispunct (int c)	prüft c ob es ein Sonderzeichen (,.-*...) ist. Rückgabewert true oder false.
int isspace (int c)	prüft c auf Whitespaces (blank, CR, FF, HT, NL, VT). Rückgabe true oder false.
int isupper (int c)	prüft c ob es ein Großbuchstabe ist. Rückgabewert true oder false.
int isxdigit (int c)	prüft c auf 0-9, a-z, A-Z. Rückgabewert true oder false.
int toupper (int c)	wenn c ein Kleinbuchstabe ist, dann wird dieser in den entsprechenden Großbuchstaben umgesetzt. Rückgabewert ist c.
int tolower (int c)	wenn c ein Großbuchstabe ist, dann wird dieser in den entsprechenden Kleinbuchstaben umgesetzt. Rückgabewert ist c.

11.1.2. math.h

double acos(double x) Arcuscosinus.
double asin(double x) Arcussinus.
double atan(double x) Arcustangens, es gibt auch noch atan2.
double ceil(double x) gibt die kleinste Ganzzahl >= x zurück.
double cos(double x) Cosinus.
double cosh(double x) Cosinus hyperbolicus.
double exp(double x) Expotentialfunktion e^x.
double fabs(double x) Absolutbetrag von x |x|.
double floor(double x) gibt die größte Ganzzahl <= x zurück.
double fmod(double x,double y) gibt den Rest von x/y (mit dem Vorzeichen von x) zurück.
double frexp(double x,int *pexp) zerlegt x in eine normalisierte Mantisse (Rückgabewert) und eine Potenz von 2, die in pexp abgespeichert wird.
double ldexp(double x, int exp) gibt x * 2^{exp} zurück.
double log(double x) natürlicher Logarithmus von x
double log10(double x) 10er Logarithmus von x.
double modf(double x,double *pint) zerlegt x in einen ganzzahligen Teil, der in pint abgespeichert wird und in einen Rest als Rückgabewert (beide mit Vorzeichen von x).
double pow(double x,double y) liefert x^y zurück.
double sin(double x) Sinus.
double sinh(double x) Sinus hyperbolicus.
double sqrt(double x) liefert die Quadratwurzel von x.
double tan(double x) Tangens.
double tanh(double x) Tangens hyperbolicus.

11.1.3. signal.h

Bekannterweise können in Unix eine Menge von Signalen an verschiedene Prozesse verschickt werden. In DOS sind das nur wenige (bei drücken von CTRL-C das Signal SIGINT, SIGFPE bei Gleitkommafehler). Normalerweise wird ein Prozeß durch das Eintreffen eines Signals abgebrochen. Mit der Funktion signal kann man eintreffende Signale ignorieren bzw. sinnvoll behandeln.

```
void (*signal( int sig, void( *func ) ( int) ) )( int );
```

11.1.4. stdio.h

Im bisherigen Skriptum wurden schon oft Funktionen der stdio verwendet. In stdio.h sind auch wichtige defines wie FILE, EOF, FOPEN_MAX, FILENAME_MAX enthalten. Hier folgt nun eine komplette Aufstellung und auch einige Beispiele.

void clearerr(FILE *f) setzt die internen Schalter für Fehler und EOF für den File f auf 0.
int fclose(FILE *f) schließen des Files f; Rückgabewert 0 wenn ok, EOF wenn Fehler.
int feof(FILE *f) wenn EOF erreicht, dann Rückgabewert true, sonst false.
int ferror(FILE *f) wenn Fehlerstatus gesetzt, dann Rückgabewert true, sonst false.
int fflush(FILE *f) schreibt Puffer auf Platte, Rückgabewert 0 wenn ok, EOF wenn Fehler.
int fgetc(FILE *f) liest nächstes Zeichen aus f ; Rückgabewert ist das Zeichen oder EOF.
int fgetpos(FILE *f,fpos_t *p) die Fileposition wird in p gespeichert; Rückgabewert 0 wenn ok.
char *fgets(char *s,int n,FILE *f) liest eine Zeile inklusive \n nach *s, wenn die Zeile aber länger als n-1 Zeichen, dann werden nur n-1 Zeichen gelesen. *s wird mit \0 abgeschlossen. Rückgabewert NULL oder s.

FILE * fopen(char *fnam,char *art) die Datei mit Name fnam wird im Modus art eröffnet. Art ist r,w,a oder r+, w+,a+ (read, write, append, + ermöglicht lesen und schreiben auf fnam). Für binäre Files wird noch ein b angehängt, wodurch keine Newline-Konvertierung stattfindet. Rückgabe ist der Filpointer oder NULL.

int fprintf(FILE *f,char *format, ...) ist ein printf auf einen File.

int fputc(int c,FILE *f) schreibt das Zeichen c in den File f. Rückgabewert Zeichen c oder EOF.

int fputs(char *s,FILE *f) schreibt String *s nach f (kein extra \n). Rückgabewert true oder EOF.

size_t fread(void *p,size_t size,size_t nelem,FILE *f) liest aus File f size*nelem Bytes nach Adresse p.Rückgabewert ist gelesene Zeichen/size, also nelem wenn ok.

FILE *freopen(char *fnam,char *art,FILE *f) schließt f und eröffnet dann die Datei mit Name fnam im Modus art. Ist schneller als getrenntes fclose, fopen .

int fscanf(FILE *f,char *format) ein scanf aus dem File f. Rückgabewert ist die Anzahl formatierter Elemente oder EOF.

int fseek(FILE *f,long anz,int art) verstellt die aktuelle Positionierung im File abhängig von art (Filebeginn, -ende, aktuelle Position) um anz Bytes bzw. auf das durch anz bezeichnete Byte (fseek(f,12,SEEK_SET) auf das 11. Byte von vorne).

int fsetpos(FILE *f,fpos_t *pos) setzt ein Positionierung im File, die früher mit fgetpos ermittelt wurde.

long ftell(FILE *f) liefert die aktuelle Positionierung (von Filebeginn) oder EOF.

size_t fwrite(void *p,size_t size,size_t nelem,FILE *f) schreibt ab Adresse p size*nelem Bytes nach File f . Rückgabewert ist nelem wenn ok oder < 0 wenn fehlerhaft.

int getc(FILE *f) gleich wie fgetc, ist aber ein Macro.

int getchar() gleich wie fgetc (stdin).

char *gets(char *s) liest eine Zeile nach *s; ersetzt \n durch \0; keine Längenbeschränkung möglich; Rückgabewert NULL oder s.

void perror(char *s) schreibt den Text der letzten internen Fehlermeldung nach stderr. Der String in *s wird getrennt durch : vor die Fehlermeldung gestellt.

int printf(char *format, ...) wurde bereits in Kapitel 3 gründlich erläutert.

int putc(int c,FILE *f) gleich wie fputc, ist aber ein Macro.

int putchar(int c) gleich wie fputc (c,stdout).

int puts(char *s) schreibt den String *s nach stdout und hängt noch ein \n an. Rückgabewert >= 0 wenn ok, sonst EOF

int remove(char *fnam) löscht die Datei mit Namen fnam. Rückgabe 0 wenn ok.

int rename(char *alt,char *neu) rename von Date mit Name *alt auf Name *neu. Rückgabe 0 wenn ok.

void rewind(FILE *f) gleich wie fseek (f,0L,SEEK_SET), positioniert auf Filebeginn.

int scanf(char *format, ...) liest formatiert aus stdin. Im Formatstring nur %-Elemente angeben. Rückgabe ist EOF oder die Anzahl erfolgreicher Formatierungen.

int sprintf(char *ziel,char *format, ...) gleich wie printf, die Ausgabe aber in den String *ziel.

int sscanf(char *von, char *format, ...) gleich wie scanf, es wird aber aus dem String *von gelesen.

FILE *tmpfile() erzeugt einen temporären File wie fopen(tname,"wb+"). Beim fclose erfolgt ein automatisches remove.

char *tmpnam(char *s) ein tauglicher Name für temporäre Dateien wird in *s abgelegt.

int ungetc(int c,FILE *f) schreibt ein Zeichen zurück in den File f, dieses wird später gelesen.

int vprintf(char *format, va_list ap) gleich wie printf, statt der Folgeparameter wird aber eine va_list verwendet (siehe stdarg.h). Es gibt auch fvprintf und svprintf.

11.1.5. stdlib.h

Hier wurden alle Funktionen gesammelt, die keine eigenen Gruppen haben.

void abort(void)	radikales Programmende, übergibt Fehler-Exit an das Betriebssystem.
int abs(int i)	Absolutwert von i.
int atexit(void (*fun) (void))	übergibt Pointer auf eine (eigene) Funktion, die beim Programmexit ausgeführt werden soll.
double atof(char *s)	wandelt die in *s stehende Konstante in einen Gleitkommatyp um.
int atoi(char *s)	wandelt die in *s stehende Konstante in einen integer Wert um.
long atol(char *s)	wandelt die in *s stehende Konstante in einen long Wert um.
void bsearch(.....)	binäres Suchen in einer Tabelle. Parameter hier nicht erklärt.
void *calloc(size_t nelem,size_t size)	belegt nelem*size Bytes Hauptspeicher für Arrays (ausgerichtet).
div_t div(int dividend,int divisor)	div_t ist eine Struktur mit Quotient und Rest.
void exit(int status)	beendet Programm ordnungsgemäß, Status an das Betriebssystem.
void free(void *p)	gibt einen vorher mit malloc belegeten Hauptspeicher wieder frei.
char *getenv(char *name)	gibt den Inhalt der Betriebsystem-Umgebungsvariable *name zurück.
long labs(long i)	Absolutwert von i (einer long Variablen).
ldiv_t ldiv(long dividend,long divisor)	wie div, aber für long Variable.
void *malloc(size_t size)	belegt size Bytes Hauptspeicher zur Programmlaufzeit. Liefert die Adresse wenn ok, sonst NULL.
void qsort (.....)	Quick-sortieren von Tabellen. Parameter hier nicht erklärt.
int rand(void)	liefert eine Pseudozufallszahl >= 0 (von 0 bis RAND_MAX).
void *realloc(void *p,size_t size)	verändert die Größe des Bereichs, der vorher mit malloc bereitgestellt wurde. Der Wert darin bleibt zumindest bei Vergrößerung erhalten.
void srand(unsigned int start)	Startwert für die mit rand gelieferte Pseudozufallszahlenkette setzen.
strtod, strtol, strtoul	konvertieren String-Konstante nach double, long, unsigned long.
int system(char *s)	In *s steht ein Betriebssystembefehl, der auf Betriebsystemniveau ausgeführt wird. Dessen Exit-Code kommt zurück.

In der C-Norm sind noch die Funktionen mblen, mbstowcs, mbtowc, wcstombs, wctomb für die Arbeit mit multibyte-Characters (2 und mehr Bytes pro Zeichen) vorgesehen.

11.1.6. string.h

Um Strings bequemer behandeln zu können ist eine größere Gruppe von Funktionen vorhanden.

void *memchr(void *s,int c, size_t n)	ein character-Array auf Adresse s, in der Länge n wird nach dem Zeichen c durchsucht. Rückgabe ist die Adresse auf der das Zeichen gefunden wurde oder NULL.
int memcmp(void *s1,void *s2, size_t n)	die Strings *s1 und *s2 werden in der Länge n verglichen. Rückgabe: *s1 > *s2 : >0, *s1 < *s2 : <0, *s1 == *s2 : 0.
void *memcpy(void *s1,void *s2, size_t n)	kopiert *s2 nach *s1 in der Länge n. Rückgabe s1 oder NULL.
void *memmove(void *s1,void *s2, size_t n)	kopiert *s2 nach *s1 in der Länge n. Rückgabe s1 oder NULL. Bei Redefinitionen ist korrektes kopieren garantiert.
char *strcat(char *s1,char *s2)	hängt *s2 an *s1 an. Rückgabe s1 oder NULL.
char *strchr(char *s1,int c)	sucht c im String *s1 (bis zum \0). Rückgabe ist die Adresse, auf der das Zeichen gefunden wurde oder NULL.
int strcmp(char *s1,char *s2)	vergleicht *s1 mit *s2 bis zum \0. Rückgabe siehe memcmp.
int strcoll(char *s1,char *s2)	vergleicht *s1 mit *s2 nach länderspezifischen Vergleichs regeln. Rückgabe siehe memcmp.
char *strcpy(char *s1,char *s2)	kopiert *s2 nach *s1 inklusive \0. Rückgabe s1 oder NULL.

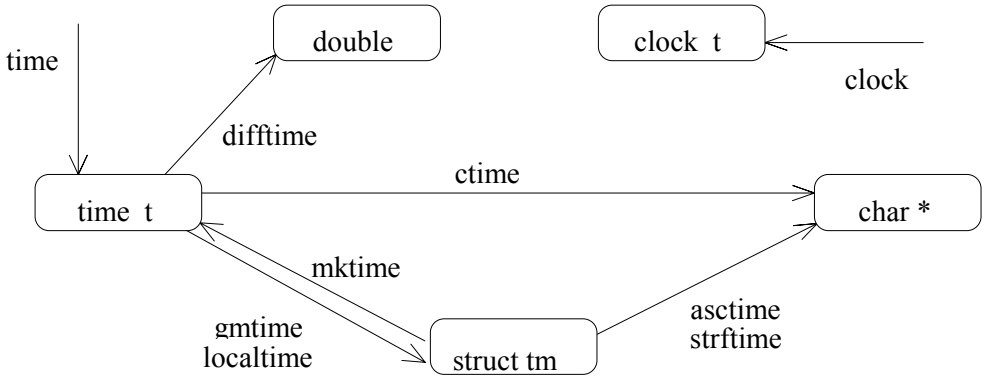
size_t strcspn(char *s1,char *s2)	sucht in *s1 nach dem ersten Zeichen, welches in *s2 vorkommt. Rückgabe ist die gefundene Suchspalte.
char *strerror(int errorcode)	liefert bei internen Fehlernummern den passenden Text .
size_t strlen(char *s)	ermittelt Länge von *s (ohne \0).
char *strncat(char *s1,char *s2, size_t n)	kopiert *s2 nach *s1 bis zu \0 oder in der maximale Länge n (sollte Länge von s1 sein). Rückgabe s1 oder NULL.
int strncmp(char *s1,char *s2, size_t n)	vergleicht *s1 mit *s2 bis zum \0 oder in der maximalen Länge n. Rückgabe siehe memcmp.
char *strncpy(char *s1,char *s2, size_t n)	kopiert *s2 nach *s1 bis zum \0 oder in der maximalen Länge n. Rückgabe s1 oder NULL.
char *strpbrk(char *s1,char *s2)	wie strcspn, Rückgabewert aber NULL oder Zeichenadresse.
char *strrchr(char *s1,int c)	wie strchr, gesucht wird aber das rechteste Zeichen gleich c.
size_t strspn(char *s1,char *s2)	wie strcspn, gesucht wird aber ein Zeichen aus *s1, welches in *s2 nicht vorkommt.
char *strstr(char *s1,char *s2)	ähnlich strchr, es wird aber geprüft, ob der gesamte String *s2 Teil des Strings *s1 ist.
char *strtok(char *s1,char *s2)	aufwendige Funktion, zerteilt in mehreren Aufrufen den String *s1 in Teilstrings (Trennung mittels Trennzeichen aus *s2).

11.1.7. time.h

Zeiten können auf verschiedene Weise dargestellt und bearbeitet werden. Als Standarddarstellung wird **time_t** eingesetzt, ein unsigned long, welcher immer eine Anzahl Sekunden enthält, die seit einem bestimmten Zeitpunkt (dem 1.1.1970) vergangen sind. Mit anderen Funktionen kann man time_t in die Darstellung der Struktur **struct tm** (siehe Kapitel über Strukturen) überleiten. Zeitwerte unter der Sekundengrenze werden in Clock-Ticks des Prozessors (seit Start des Programms) ermittelt. CLOCKS_PER_SEC sagt aus wieviele Clock-Ticks eine Sekunde ergeben. Am PC ist dies meist 1000.

char *asctime(struct tm *zp)	die Zeit aus *zp wird als String mit Standardformat formatiert.
clock_t clock()	Rückgabewert: die Clock-Ticks seit Programmstart oder -1.
char *ctime(time_t *zs)	Die Zeit aus *zs wird als String mit Standardformat formatiert.
double difftime(time_t t1,time_t t2)	Zeitdifferenz t1 - t2 in Sekunden.
struct tm *gmtime(time_t *zs)	Die Teit aus *zs wird in einer Struktur tm (für die Zeitzone UTC = Zeit in Greenwich) aufbereitet und deren Adresse (oder NULL) retourniert.
struct tm *localtime(time_t *zs)	genauso wie gmtime, aber für die lokale Zeitzone (muß auch gesetzt sein, sonst eine amerikanische Zeitzone).
time_t mktime(struct tm *zp)	Wandelt Zeitformat struct tm nach Format time_t um.
size_t strftime(char *s, size_t n, char *format, struct tm *zp)	die Zeit aus *zp wird gemäß dem Formatierstring *format (ähnlich printf, aber zahlreiche zeitspezifische Formatelemente für Minuten, etc.) aufbereitet und in *s bis zur maximalen Länge von n-1 abgespeichert. Rückgabewert ist Länge von *s oder 0.
time_t time(time_t *zs)	die aktuelle Maschinenzeit wird im Zeitformat time_t in *zs abgelegt (wenn zs != NULL) und auch zurückgegeben (oder -1 bei Fehler).

Zum besseren Verständnis noch eine Überblicksdarstellung der Funktionen:



11.2. Grafikmöglichkeiten unter DOS mit Borland BGI

Turbo C++ verfügt über die Bibliothek GRAPHICS.LIB mit über 70 Grafikfunktionen für Linien, Figuren, Flächenfüllung, Schriftarten, etc., die in der Datei **graphics.h** als Prototypen deklariert sind.

Der Bildschirm unter DOS funktioniert entweder im Text- oder im Grafikmode. Als erstes muß der Schirm (die Grafikkarte) in den Grafikmode (EGA, VGA) umgeschaltet werden.

int graphdriver, graphmode, errorcode;

graphdriver = DETECT; die Bestimmung des installierten Grafik-Adapters (EGA, VGA) geschieht somit automatisch.

initgraph(&graphdriver,&graphmode,"c:\\tc\\bgi"); prüft die Hardware, schaltet in den Grafikmodus und lädt das entsprechende Treiberprogramm (EGAVGA.BGI) aus dem angegebenen Verzeichnis (3. Parameter). Ein Leerstring bedeutet aktuelles Verzeichnis.

Die Funktion **graphresult** liefert den Fehlercode der letzten Grafikoperation und setzt den Fehlerstatus auf grOk zurück. Fehlerfrei: Ergebniscode = 0 (Konstante grOk = 0).

```
errorcode = graphresult();                    // speichern von graphresult
if (errorcode != grOk)
{ printf("Grafik - Fehler : %s\n", grapherrormsg(errorcode));
  printf("Programm abgebrochen");
  getch();
  exit (1);             // Rücksprung zum Betriebssystem mit Fehlermeldung
}
```

cleardevice löscht den Bildschirm.

closegraph beendet den Grafikmodus und stellt den Textmodus wieder her.

Innerhalb des Programms kann mit **restorecrtmode** und **setgraphmode** beliebig zwischen Text- und Grafikmodus umgeschaltet werden.

Verschiedenste grafische Ausgabefunktionen verwenden voreingestellte Vordergrund- und Hintergrundfarben. Die Zeichenfarbe (Vordergrund) wird mit **setcolor(color)**, die Hintergrundfarbe mit **setbkcolor(color)** eingestellt.

Während man am Textbildschirm meistens 24(25) Zeilen und 80 Spalten vorfindet hat der Grafikmodus keinerlei zeichenorientierte Einteilung. Zeichnerische und auch textmäßige Ausgaben im Grafikmodus verwenden den **Grafik-Cursor**, der auf einem Bildschirmpixel (gemäß Auflösung, normalerweise VGA = 640 x 480) steht und nicht sichtbar ist.

moveto(x,y) setzt den Grafik-Cursor auf den angegebenen Punkt im 640 x 480 Bereich (bedenken Sie, daß EGA oder Super VGA andere Auflösungen hat). "Clipping" beeinflußt diese Positionierung nicht.

getx/y liefert die momentane X/Y-Koordinaten des Grafik-Cursors.

getmaxx/y ermittelt die maximal mögliche X/Y-Koordinate des Bildschirms.

Das **Koordinatensystem** hat seinen Ursprung in der linken obere Ecke des Bildschirms. Bei einer Bildschirmauflösung von 640 x 480 Punkten ergeben sich folgende Koordinaten : Ecke links oben (0,0) rechts oben (639,0) links unten (0,479) rechts unten (639,479).

Die **Textausgabe** in simplen grafischen Systemen wie dem BGI verwendet einen bitweise definierten Zeichensatz (8*8 Punkte pro Zeichen) und mehrere Vektor-Zeichensätze (.CHR), die als normale Dos-Dateien vorliegen.

Die Ausgabe wird entweder mit **outtext("string")** oder **outtextxy(x,y;"string")** vorgenommen. Ist "Clipping" nicht aktiv, so werden Ausgaben mit dem Standardzeichensatz komplett unterdrückt, wenn rechts von der momentanen Grafikcursorposition (bzw. x,y) nicht genügend Platz vorhanden ist (die Zeichengröße der gewählten Schrift bestimmt den Platzbedarf !

Vor der Ausgabe können mit den Funktionen **settextstyle(font,direction,charsize)** und **settextjustify(horiz,vert)** noch Schriftart, -richtung, -größe und die Ausrichtung (li, zentriert, re; unten, z, oben) eingestellt werden.

Normalerweise bezieht sich die Ausgabe auf alle Pixel des Bildschirms. Es ist jedoch möglich ein **Grafikfenster** zu definieren. Mit **setviewport(x1,y1,x2,y2,clip)** läßt sich ein rechteckiges Fenster am Bildschirm festlegen. Der Grafik-Cursor geht automatisch auf den Ursprung dieses Fensters. Die folgenden Ausgaben werden sich dann auf dieses Fenster beschränken, auch der Ursprung des Koordinatensystems liegt links oben im Fenster. Der Bildschirminhalt des Fensters bleibt erhalten, kann aber mit **clearviewport** gelöscht werden.

Mit **putpixel(x,y;color)** kann ein Punkt in der durch color festgelegten Farbe gezeichnet werden.

getpixel(x,y) liefert die Farbe eines Pixels zurück.

Objekte und Zeichenstil

line(x1,y1,x2,y2) zeichnet eine Linie von - nach, der Grafik-Cursor bleibt unverändet.

lineto(x,y) Linie von der momentanen Position weg

rectangle(x1,y1,x2,y2) Rechteck

bar(x1,y1,x2,y2) gefülltes Rechteck

arc(x,y;start,ende,radius) Kreisausschnitt

circle(x,y;radius) Kreis

ellipse(x,y;start,ende,xradius,yradius) Ellipse

fillellipse(x,y;xradius,yradius) ausgefüllter (elliptischer) Kreis

drawpoly(numpoints,polypoints) Umriß eines Vielecks

Für diese Objekte können **vorher** mit **setlinestyle(linestyle,pattern,thickness)** die Linienart und mit **setfillstyle(pattern,color)** das Muster der Flächenfüllung gewählt werden.

floodfill(x,y;border) füllt einen mit border umschlossenen Bereich mit dem gesetzten Füllmuster.

Übungen :

- Zeichnen Sie ein Netz von senkrechten und waagrechten Linien zur Einstellung des Monitors auf den Bildschirm. Die Strichstärke soll abwechselnd normal und dick, die Farben sollen unterschiedlich sein.

- Geben Sie Ihren Namen und Ihre Adresse doppelt eingerahmt in der Mitte des Bildschirms aus. Verwenden Sie verschiedene Farben, Größen und Linienarten für Schrift, Rahmen und Hintergrund.

- Geben Sie die Monatsumsätze eines Jahres als vertikales Balkendiagramm aus. Der Mittelwert wird als horizontale Linie eingezeichnet. Neben der Ordinate soll der maximale Monatsumsatz und der Mittelwert in Zahlen stehen.

- Aus dem Mathematikbuch wissen Sie wie eine Sinuskurve im Bereich von 0 bis 2 Pi aussieht. Zeichen Sie diese formatfüllend auf den Bildschirm.